Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers

Thurston H.Y. Dang
University of California, Berkeley

Petros Maniatis
Google Brain

David Wagner
University of California, Berkeley

Abstract

Using memory after it has been freed opens programs up to both data and control-flow exploits. Recent work on temporal memory safety has focused on using explicit lock-and-key mechanisms (objects are assigned a new lock upon allocation, and pointers must have the correct key to be dereferenced) or corrupting the pointer values upon `free()`. Placing objects on separate pages and using page permissions to enforce safety is an older, well-known technique that has been maligned as too slow, without comprehensive analysis. We show that both old and new techniques are conceptually instances of lock-and-key, and argue that, in principle, page permissions should be the most desirable approach. We then validate this insight experimentally by designing, implementing, and evaluating Oscar, a new protection scheme based on page permissions. Unlike prior attempts, Oscar does not require source code, is compatible with standard and custom memory allocators, and works correctly with programs that `fork`. Also, Oscar performs favorably – often by more than an order of magnitude – compared to recent proposals: overall, it has similar or lower runtime overhead, and lower memory overhead than competing systems.

1 Introduction

A temporal memory error occurs when code uses memory that was allocated, but since freed (and therefore possibly in use for another object), i.e., when an object is accessed outside of the time during which it was allocated.

Suppose we have a function pointer stored on the heap that points to function `Elmo()` (see Figure 1) at address `0x05CADA`. The pointer is used for a bit and then de-allocated. However, because of a bug, the program accesses that pointer again after its deallocation.

This bug creates a control-flow vulnerability. For example, between the de-allocation (line 7) and faulty re-use of the pointer (line 10), some other code could allocate the same memory and fill it from an untrusted source – say a network socket. When the de-allocated pointer is faultily invoked, the program will jump to whatever address is stored there, say the address of the ROP gadget `Grouch()` at address `0x05DEAD`, hijacking control flow.

```
1  void(**someFuncPtr)() = malloc(sizeof(void*));
2  *someFuncPtr = &Elmo;     // At 0x05CADA
3  (*someFuncPtr)();         // Correct use.
4  void(**callback)();
5  callback = someFuncPtr;
6  ...
7  free(someFuncPtr);        // Free space.
8  userName = malloc(...); // Reallocate space.
9  ... // Overwrite with &Grouch at 0x05DEAD.
10 (*callback)();            // Use after free!
```

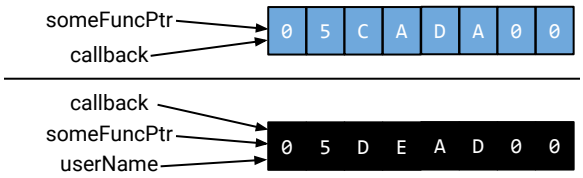

Figure 1: Top: `someFuncPtr` and `callback` refer to the function pointer, stored on the heap. Bottom: `userName` reuses the freed memory, formerly of `someFuncPtr/callback`.

Heap temporal memory safety errors are becoming increasingly important [27, 42]. Stack-allocated variables are easier to protect, e.g., via escape analysis, which statically checks that pointers to a stack variable do not outlive the enclosing stack frame, or can be reduced to the heap problem, by converting stack allocations to heap allocations [33]. Stack use-after-free is considered rare [42] or difficult to exploit [27]; a 2012 study did not find any such vulnerabilities in the CVE database [15]. We therefore focus on temporal memory safety for heap-

allocated objects in the rest of this paper.

Various defenses have been tried. A decade ago, Dhurjati and Adve [23] proposed using page permissions and aliased virtual pages for protection. In their scheme, the allocator places each allocated object on a distinct virtual page, even though different objects may share the same physical page; when an object is deallocated, the corresponding virtual page is rendered inaccessible, causing pointer accesses after deallocation to fail. Although a combination of the technique with static analysis led to reasonable memory economy and performance, critics found faults with evaluation and generality, and – without quantitative comparison – summarily dismissed the general approach as impractical [31, 42], or without even mentioning it [41]. Since then, researchers have proposed more elaborate techniques (CETS [31], DangSan [41], Dangling Pointer Nullification [27] ("DangNull") and FreeSentry [42]), relying on combinations of deeper static analysis and comprehensive instrumentation of heap operations such as object allocation, access, and pointer arithmetic. However, these schemes have yielded mixed results, including poor performance, partial protection, and incompatibility.

In this work, we first study past solutions, which we cast as realizations of a *lock-and-key* protection scheme (Section 2). We argue that using page permissions to protect from dangling pointers, an implicit lock-and-key scheme with lock changes, is less brittle and complex, and has the potential for superior performance. We then develop *Oscar*, a new protection mechanism using page permissions, inspired by Dhurjati and Adve's seminal work [23]. We make the following contributions:

- We study in detail the overhead contributed by the distinct factors of the scheme – shared memory mappings, memory-protection system calls invoked during allocation and deallocation, and more page table entries and virtual memory areas – using the standard SPEC CPU 2006 benchmarks (Section 3).
- We reduce the impact of system calls by careful amortization of virtual-memory operations, and management of the virtul address space (Section 4).
- We extend Oscar to handle server workloads, by supporting programs that `fork` children and the common case of custom memory allocators other than those in the standard C library (Section 5).
- We evaluate Oscar experimentally using both SPEC CPU 2006 and the popular memcached service, showing that Oscar achieves superior performance, while providing more comprehensive protection than prior approaches.

Our work shows, in principle and experimentally, that protection based on page permissions – previously thought to be an impractical solution – may be the most promising for temporal memory safety. The simplicity of the scheme leads to excellent compatibility, deployability, and the lowest overhead: for example, on SPEC CPU, CETS and FreeSentry have 48% and 30% runtime overhead on `hmmer` respectively, vs. our 0.7% overhead; on `povray`, DangNull has 280% overhead while ours is $< 5\%$. While DangSan has runtime overhead similar to Oscar, DangSan's memory overhead (140%) is higher than Oscar's (61.5%). Also, our study of `memcached` shows that both standard and custom allocators can be addressed effectively and with reasonable performance.

2 Lock-and-Key Schemes

Use of memory after it has been freed can be seen as an authorization problem: pointers grant access to an allocated memory area and once that area is no longer allocated, the pointers should no longer grant access to it. Some have therefore used a lock-and-key metaphor to describe the problem of temporal memory safety [31]. In this section, we show how different published schemes map to this metaphor, explicitly and sometimes implicitly, and we argue that page-permission-based protection may be the most promising approach for many workloads (see Table 1 for a summary).

2.1 Explicit Lock-and-Key: Change the Lock

In this scheme, each memory allocation is assigned a lock, and each valid pointer to that allocation is assigned the matching key. In Figure 1, the code is modified so in line 1, the allocated object gets a new lock (say 42), and the matching key is linked to the pointer (see Figure 2). Similarly, in line 5, the key linked to `someFuncPtr` is copied to `callback`. The code is instrumented so that pointer dereferencing (lines 3 and 10) is preceded by a check that the pointer's key matches the object's lock.

When the space is deallocated and reallocated to a new object, the new object is given a new lock (say, 43), and `userName` receives the appropriate key in line 8. The keys for `someFuncPtr` and `callback` no longer match the lock past line 7, avoiding use after free (Figure 3).

Since this scheme creates explicit keys (one per pointer), the memory overhead is proportional to the number of pointers. The scheme also creates one lock per object, but the number of objects is dominated by the number of pointers.

Example Systems: Compiler-Enforced Temporal Safety for C (CETS) [31] is an example of this scheme. Although in our figure we have placed the key next to the pointer (similar to bounds-checking schemes that store

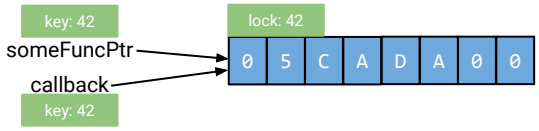

Figure 2: Each pointer has a key, each object has a lock.

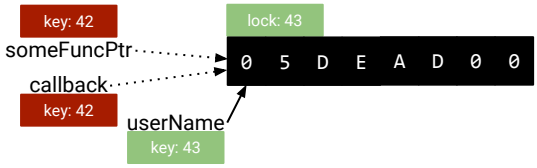

Figure 3: Lock change (see Figure 2 for the 'Before').

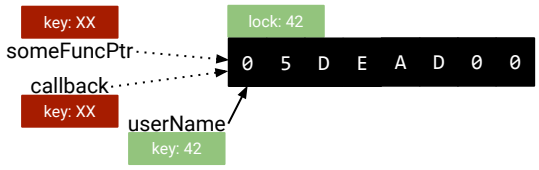

Figure 4: Key revocation (see Figure 2 for the 'Before').

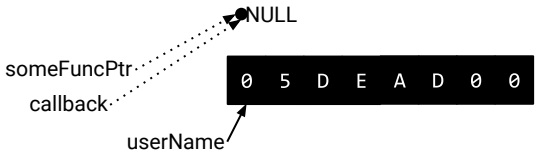

Figure 5: After pointer nullification (see Figure 1 for the 'Before'), object space can be reused safely.

both the pointer plus the size [25], called *plus-size pointers*) and lock next to the object, this need not be the case in implementations. Indeed, one of the key advances of CETS over prior lock-and-key schemes is that it uses a disjoint metadata space, with a separate entry for each pointer that stores the key and the lock location; this avoids changing the memory layout of the program.

2.2 Explicit Lock-and-Key: Revoke the Keys

Instead of changing the lock, one could revoke all keys upon reallocation. This requires tracking of keys throughout memory; for example, freeing either `someFuncPtr` *or* `callback` should revoke the keys for both pointers (Figure 4).

To enable this, upon allocation (line 1) instrumentation must maintain global metadata tracking all pointers to a given object, and this index must be updated at every relevant assignment (line 5). Deallocation (line 7) must be followed by looking up all pointers to that object, revoking (nullifying or otherwise invalidating) their keys. Revoking keys is harder than changing the lock, since it requires tracking of key propagation.

Example Systems: To our knowledge, this has not been used for any published explicit lock-and-key scheme; but, it segues to the next idea that has been used in prior work: revoking the keys with *implicit* lock-and-key.

2.3 Implicit Lock-and-Key: Revoke the Keys

We can view a pointer as the key, and the object as the lock. Thus, instead of revoking a key from a separate *explicit* namespace, we can change the pointer's value [27].

The relevant code instrumentation is similar to the explicit case. Upon allocation or pointer assignment, we update a global index tracking all pointers to each object. Upon deallocation, we find and corrupt the value of all pointers to the deallocated object (Figure 5), say by setting them to `NULL`. Pointer dereferences need not be instrumented, since the memory management unit (MMU) performs the null check in hardware.

Although this scheme does not need to allocate memory for explicit lock or key fields, it does need to track the location of each pointer, which means the physical memory overhead is at least proportional to the number of pointers.[1]

Example Systems: DangNull's dangling pointer nullification [27] is an example of this scheme. FreeSentry [42] is similar, but instead of nullifying the address, it flips the top bits, for compatibility reasons (see Section 6.3). DangSan [41] is the latest embodiment of this technique; its main innovation is the use of append-only per-thread logs for pointer tracking, to improve runtime performance for multi-threaded applications.

2.4 Implicit Lock-and-Key: Change the Lock

Implicit lock-and-key requires less instrumentation than explicit lock-and-key, and changing locks is simpler than tracking and revoking keys. The ideal scheme would therefore be implicit lock-and-key in which locks are changed.

One option is to view the object as a lock, but this lacks a mechanism to "change the lock". Instead, it is more helpful to view the *virtual address* as the lock.

[1] DangSan can use substantially more memory in some cases due to its log-based design.

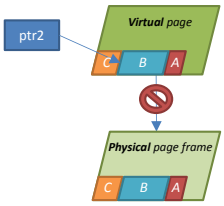

Figure 6: The virtual page has been made inaccessible: accesses to objects A, B or C would cause a fault.

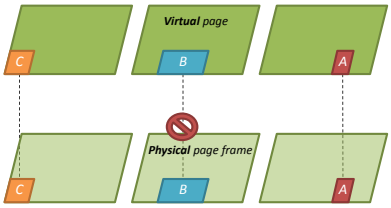

Figure 7: With one object per page, we can selectively disable object B.

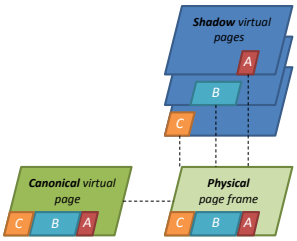

Figure 8: Each object has its own shadow virtual page, which all map to the same physical frame.

Recall that objects (and physical memory) are accessed via virtual addresses, which are translated (by the MMU) into physical addresses. By removing the mapping or changing the page permissions, we can make a virtual page inaccessible; the underlying physical memory can then be mapped to a different virtual address (changed lock) for reuse. A drawback is that making a virtual page inaccessible renders *all* objects on that page – often a non-trivial number, since pages are 4KB or larger – inaccessible (Figure 6). Placing one object per page (Figure 7) is wasteful of memory resources: it uses more memory and strains the cache and the TLB.

It is not strictly necessary to use page permissions to enforce page inaccessibility after deallocation. In principle, we could maintain a hashtable of live pointers, and instrument all the pointer dereferences to check that the pointer is still live, trading off instrumentation for system calls. This would still have less overhead than an explicit lock-and-key scheme, because we would not need to instrument pointer arithmetic.

Example Systems: Electric Fence [9] implements this scheme, by placing one object per physical frame. Its high physical memory usage renders it impractical for anything other than debugging.

Dhurjati and Adve [23] overcame this shortcoming through virtual aliasing. Normally, `malloc` might place multiple objects on one virtual page, which Dhurjati and Adve refer to as the *canonical* virtual page. For each object on the canonical virtual page, they create a *shadow* virtual page that is aliased onto the same underlying physical page frame. This allows each object to be disabled independently (by changing the permissions for the corresponding shadow page), while using physical memory/cache more efficiently than Electric Fence (Figure 8). However, this still requires many syscalls and increases TLB pressure. Furthermore, creating shadows introduces compatibility issues with `fork` (Section 5.1).

The physical memory overhead – one page table entry, one kernel virtual memory area struct, plus some user-space allocator metadata, per object – is proportional to the number of live objects. We expect this to be more efficient than the other classes of lock-and-key schemes, which have overhead proportional to the number of pointers (albeit with a smaller constant factor). Some engineering is required to avoid stateholding of `munmap`'ed page table entries (Section 8).

2.5 Summary of Lock and Key Schemes

Table 1 compares the plausible lock-and-key schemes. Implicit lock-and-key schemes that change the lock (i.e., one object per virtual page) are advantageous by having no overhead for any pointer arithmetic, and no direct cost (barring TLB and memory pressure) for pointer dereferences. Furthermore, the core technique does not require application source code: for programs using the standard allocator, we need only change the `glibc` `malloc` and `free` functions. However, Dhurjati and Adve's full scheme requires application source code to apply their static analysis optimization, which allows them to reuse virtual addresses when a pool is destroyed.

3 Baseline Oscar Design

We will develop the shadow virtual pages idea in a direction that does not require source-code analysis, with less stateholding of kernel metadata for freed objects, and with better compatibility with `fork`. We focus on `glibc` and Linux.

While we have argued that page-permissions-based protections should require less instrumentation than newer schemes, there has been no good data on the overhead of shadows (without reliance on static analysis), let alone quantitative comparisons with recent schemes. In the first part of this paper, we quantify and predict the overhead when using only shadows. These measurements informed our approach for reducing the overhead, which are described in the second part of this paper.

To help us improve the performance of shadow-page-based schemes, we first measure their costs and break

Instrumentation	Explicit lock-and-key: changing the lock e.g., CETS	Implicit lock-and-key: revoking the keys e.g., DangNull/FreeSentry	Implicit lock-and-key: changing the lock e.g., Electric Fence
`malloc ()`	Allocate lock address; Issue key; Set lock	Register pointer	Syscall to create virtual page
Simple ptr arithmetic: `p+=2`	✓ No cost		
General ptr arithmetic: `p=q+1`	Propagate lock address and key	Update ptr registration	✓ No cost
Pointer dereference: `*p`	Check key vs. lock value (at lock address)	✓ No cost	<TLB and memory pressure>
`free ()`	Deallocate lock address	Invalidate pointers	Syscall to disable virtual page
No application source needed	Needs source + recompilation		✓ Yes; Req'd by Dhurjati&Adve
Physical memory overhead	O(# pointers)	O(# pointers)	✓ O(# objects)

Table 1: Comparison of lock-and-key schemes. Green and a tick indicates an advantageous distinction.

down the source of overhead. Shadow-page schemes consist of four elements: modifying the memory allocation method to allow aliased virtual pages, inline metadata to record the association between shadow and canonical pages, syscalls to create and disable shadow pages, and TLB pressure. We measure how much each contributes to the overhead, so we can separate out the cost of each.

It is natural to hypothesize that syscall overhead should be proportional to the number of `malloc`/`free` operations, as page-permissions-based schemes add one or two syscalls per `malloc` and `free`. However, the other costs (TLB pressure, etc.) are less predictable, so measurements are needed.

Our baseline design [23] uses inline metadata to let us map from an object's shadow address to its canonical address. When the program invokes `malloc(numBytes)`, we allocate instead with `internal_malloc(numBytes + sizeof(void*))` to allocate an object within a physical page frame and then immediately perform a syscall to create a shadow page for the object. The object's canonical address is stored as inline metadata within the additional `sizeof(void*)` bytes. This use of inline metadata is transparent to the application, unlike with plus-size pointers. Conceivably, the canonical addresses could instead be placed in a disjoint metadata store (similar to CETS), improving compactness of allocated objects and possibly cache utilization, but we have not explored this direction.

3.1 Measurement Methodology

We quantified the overhead by building and measuring incrementally more complex schemes that bridge the design gap from `glibc`'s `malloc` to one with shadow virtual pages, one overhead factor at a time.

Our first scheme simply changes the memory allocation method. As background, `malloc` normally obtains large blocks of memory with the `sbrk` syscall (via the macro `MORECORE`), and subdivides it into individual objects. If `sbrk` fails, `malloc` obtains large blocks using `mmap(MAP_PRIVATE)`. (This fallback use of `mmap` should not be confused with `malloc`'s special case of placing very large objects on their own pages.) We cannot create shadows aliased to memory that was allocated with either `sbrk` or `mmap(MAP_PRIVATE)`; the Linux kernel does not support this. Thus, our first change was MAP_SHARED arenas: we modified `malloc` to always obtain memory via `mmap(MAP_SHARED)` (which can be used for shadows) instead of `sbrk`. This change unfortunately affects the semantics of the program if it `fork()`s: the parent and child will share the physical page frames underlying the objects, hence writes to the object by either process will be visible to the other. We address this issue – which was not discussed in prior work – in Section 5.1.

MAP_SHARED with padding further changes `malloc` to enlarge each allocation by `sizeof(void*)` bytes for the canonical address. We do not read or write from the padding space, as the goal is simply to measure the reduced locality of reference.

Create/disable shadows creates and disables shadow pages in the `malloc` and `free` functions using `mremap` and `mprotect(PROT_NONE)` respectively, but does not access memory via the shadow addresses; the canonical address is still returned to the caller. To enable the `free` function to disable the shadow page, we stored the shadow address inside the inline metadata field (recall that in the complete scheme, this stores the canonical).

Use shadows returned shadow addresses to the user. The *canonical* address is stored inside the inline metadata field. This version is a basic reimplementation of a shadow-page scheme.

All timings were run on Ubuntu 14.04 (64-bit), using an Intel Xeon X5680 with 12GB of RAM. We disabled hyper-threading and TurboBoost, for more consistent timings. Our "vanilla" `malloc`/`free` was from `glibc 2.21`. We compiled the non-Fortran SPEC CPU2006 benchmarks using `gcc/g++` v4.8.4 with `-O3`. We configured `libstdc++` with `--enable-libstdcxx-allocator=malloc`, and configured the kernel at run-time to allow more virtual memory mappings.

We counted `malloc` and `free` operations using

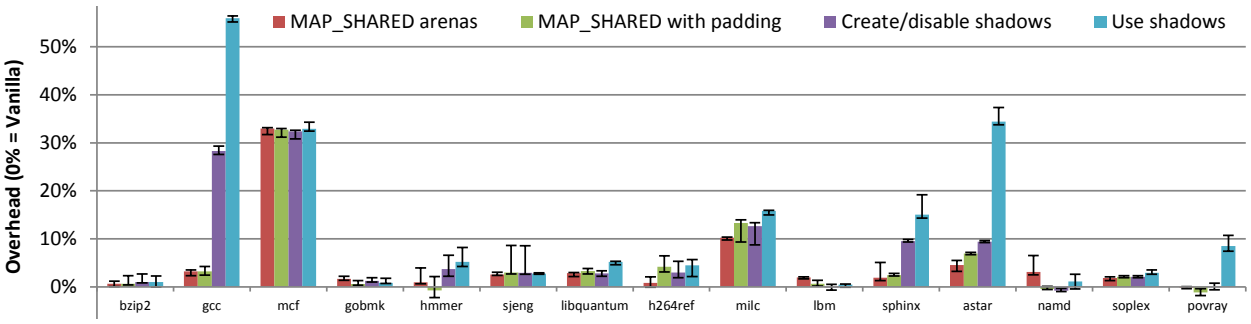

Figure 9: SPEC CPU2006 C/C++ benchmarks, showing the overhead as we reach the full design.

`mtrace`. We placed `mtrace` at the start of `main`, which does miss a small number of allocations (e.g., static initializers and constructors for global C++ objects), but these are insignificant.

3.2 Results

The overhead measurements of the four incrementally more complete schemes are shown in Figure 9 for 15 of the 19 SPEC CPU2006 C/C++ benchmarks. The remaining four benchmarks (`perlbench`, `dealII`, `omnetpp`, `xalancbmk`) exhaust the physical memory on the machine when creating/disabling shadows, due to the accumulation of `vm_area_structs` corresponding to `mprotect`'ed pages of "freed" objects. We therefore defer discussion of them until the following section, which introduces our improvements to the baseline design.

Even for the complete but unoptimized scheme (Use shadows), most benchmarks have low overhead. `gcc` and `sphinx` have high overhead due to creating/destroying shadows, as well as using shadows. `astar` and `povray` have a noticeable cost mainly due to using shadows, a cost which is not present when merely creating/disabling shadows; we infer that the difference is due to TLB pressure. Notably, `mcf`'s overhead is entirely due to MAP_SHARED arenas, as is most of `milc`'s. Inline padding is a negligible cost for all benchmarks.

In Figure 10, we plot the run-time of creating/disabling shadows, against the number of shadow-page-related syscalls[2]. We calculated the y-values by measuring the runtime of Create/disable shadows (we used the high watermark optimization from Section 4 to ensure all benchmarks complete) *minus* MAP_SHARED with padding: this discounts runtime that is not associated with syscalls for shadows. The high correlation matches our mental model that each syscall has an approximately fixed cost, though it is clear from `omnetpp` and `perlbench` that it is not perfectly fixed. Also, we can

[2]A `realloc` operation involves both creating a shadow and destroying a shadow, hence the number of `malloc/free` operations is augmented with `(2 * realloc)`.

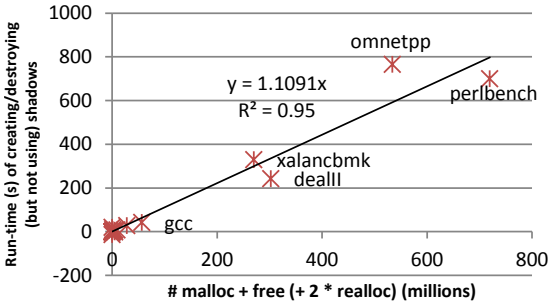

Figure 10: Predicting syscall overhead.

see that `perlbench`, `dealII`, `omnetpp` and `xalancbmk` each create over 100 million objects, which is why they could not run to completion using the unoptimized implementation.

4 Lowering Overhead Of Shadows

The previous section shows that the overhead is due to `MAP_SHARED`, creating/destroying shadows, and using shadows. The cost of using shadows – via TLB pressure – can be reduced with hardware improvements, such as larger TLBs (see Section 6.2). In this section, we propose, implement, and measure three optimizations for reducing the first two costs.

High water mark. The naïve approach creates shadows using `mremap` without a specified address and disables shadows using `mprotect(PROT_NONE)`. Since disabled shadows still occupy virtual address space, new shadows will not reuse the addresses of old shadows, thus preventing use-after-free of old shadows. However, the Linux kernel maintains internal data structures for these shadows, called `vm_area_structs`, consuming 192 bytes of kernel memory per shadow. The accumulation of `vm_area_structs` for old shadows prevented a few benchmarks (and likely many real-world applications) from running to completion.

We introduce a simple solution. Contrary to conven-

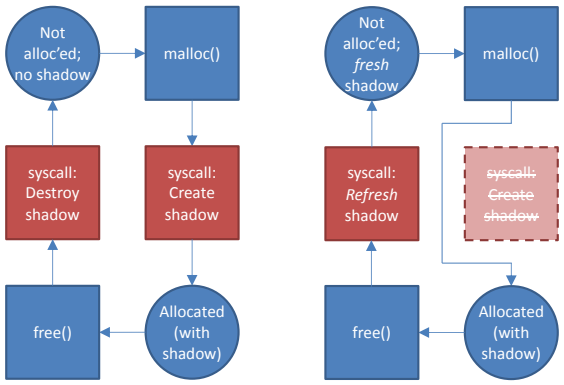

Figure 11: Left: Simplified lifecycle of a chunk of memory. Right: The `destroyShadow` syscall has been modified to simultaneously destroy the old shadow and create a new one.

tional wisdom [23], with a small design modification, Oscar can both unmap and prevent reuse of a virtual page. We use a "high water mark" for shadow addresses: when Oscar creates a shadow, we specify the high water mark as the requested shadow address, and then increment the high water mark by the size of the allocation. This is similar to the `sbrk` limit of `malloc`. Oscar can now safely use `munmap` to disable shadows, without risk of reusing old shadows. As we show in Section 6.1, virtual address space exhaustion is an unlikely, tractable problem.

Our scheme, including the high water mark, is compatible with address space layout randomization (ASLR). At startup, we initialize the high-water mark at a fixed offset to the (randomized) heap base address. To reduce variability in run-times, all benchmarks, including the baseline, were measured without ASLR, as is typical in similar research [40].

Refreshing shadows. Figure 11 (left) depicts the simplified circle of life of a heap-allocated chunk of physical memory. Over the lifetime of a program, that chunk may be allocated, freed, allocated, freed, etc., resulting in syscalls to create a shadow, destroy a shadow, create a shadow, destroy a shadow, etc. Except for the very first time a chunk has been created by `malloc`, every shadow creation is preceded by destroying a shadow.

Oscar therefore speculatively creates a new shadow each time it destroys a shadow, in Figure 11 (right). This saves the cost of creating a new shadow, the next time an object is allocated on that canonical page. The optimistically renewed shadow is stored in a hash table, keyed by the size of shadow (in number of pages) and the address of the canonical *page* (not the canonical object). This means the shadow address can be used for the next similarly-sized object allocated on the canonical page(s), even if the new object does not coincide precisely with the old object's size or offset within the page. It also improves the likelihood that the shadow can be used when objects are coalesced or split by the allocator.

Up to now, we have used `mremap` to create shadows. `mremap` actually can be used to both destroy an old mapping and create a new virtual address mapping (at a specified address) in a single system call. We use this ability to both destroy the old shadow mapping and create a new one (i.e., refresh a shadow) with one system call, thereby collapsing 2 system calls to 1 system call. This optimization depends on the high water mark optimization: if we called `mremap` with `old_size = new_size` without specifying a `new_address`, `mremap` would conclude that there is no need to change the mappings at all, and would return the old shadow virtual address.

Using MAP_PRIVATE when possible. As mentioned earlier, `MAP_SHARED` is required for creating shadows, but sometimes has non-trivial costs. However, for large objects that `malloc` places on their own physical page frames, Oscar does not need more than one shadow per page frame. For these large allocations, Oscar uses `MAP_PRIVATE` mappings.

Implementing `realloc` correctly requires care. Our ordinary `realloc_wrapper` is, in pseudo-code:

```
munmap(old_shadow);
new_canonical = internal_realloc(old_canonical);
new_shadow = create_shadow(new_canonical);
```

This works when all memory is `MAP_SHARED`. However, if the reallocated object (`new_canonical`) is large enough to be stored on its own `MAP_PRIVATE` pages, `create_shadow` will allocate a different set of physical page frames instead of creating an alias. This requires copying the contents of the object to the new page frames. Copying is mildly inefficient, but few programs use `realloc` extensively.

The overhead saving is upper-bounded by the original cost of MAP_SHARED arenas.

Abandoned approach: Batching system calls. We tried batching the creation or destruction of shadows, but did not end up using this approach in Oscar.

We implemented a custom syscall (loadable kernel module ioctl) to create or destroy a batch of shadows. When we have no more shadows for a canonical page, we call our `batchCreateShadow` ioctl once to create 100 shadows, reducing the amortized context switch cost per malloc by 100x. However, this does not reduce the overall syscall cost by 100x, since `mremap`'s internals are costly. In a microbenchmark, creating and destroying 100 million shadows took roughly 90 seconds with individual `mremap`/`munmap` calls (i.e., 200 million syscalls) vs. ≈80 seconds with our batched syscall. The savings of 10 seconds was consistent with the time to call a no-op ioctl 200 million times.

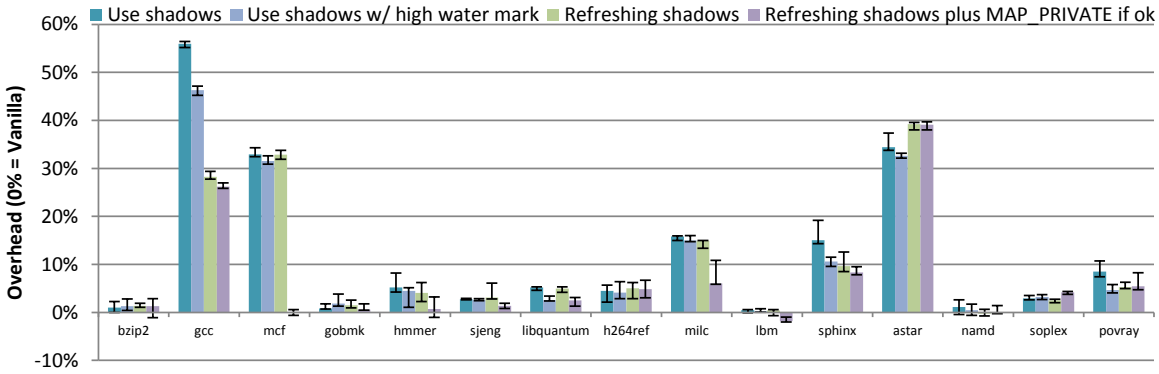

Figure 12: SPEC CPU2006 C/C++ benchmarks, showing the benefits of our optimizations.

In our pilot study, batching did not have a significant benefit. It even slowed down some benchmarks, due to mispredicting which shadows will be needed in the future. For example, we may create 100 shadows for a page that contains solely of a single object which is never freed, wasting 99 shadows.

We also tried batch-disabling shadows: any objects that are `free()`'d are stored in a "quarantine" of 100 objects, and when the quarantine becomes full, we disable all 100 shadows with a single batched syscall, then actually free those 100 objects. This approach maintains temporal memory safety, unlike the standard use of quarantine (see Section 7). Unlike batch-creating shadows, with batch-deletion we need not predict the future.

In our pilot study, batch deletion had mixed effects on runtime overhead. We hypothesize this is due to disrupting favorable memory reuse patterns: `malloc` prefers to reuse recently freed objects, which are likely to be hot in cache; quarantine prevents this.

4.1 Performance Evaluation

The effect of these improvements on the previous subset of 15 benchmarks is shown in Figure 12.

Our first two optimizations (high water mark, refreshing shadows) greatly reduce the overhead for `gcc` and `sphinx`; this is not a surprise, as we saw from Figure 9 that much of `gcc` and `sphinx`'s overhead is due to creating/destroying shadows. These two optimizations do not benefit `mcf`, as its overhead was entirely due to MAP_SHARED arenas; instead, fortuitously, the overhead is eliminated by the `MAP_PRIVATE` optimization. The `MAP_PRIVATE` optimization also reduces the overhead on `milc` by roughly ten percentage points, almost eliminating the overhead attributed to `MAP_SHARED`.

The four allocation-intensive benchmarks are shown in Figure 13. Recall that for these benchmarks, the baseline scheme could not run to completion, owing to the excessive number of leftover `vm_area_structs`

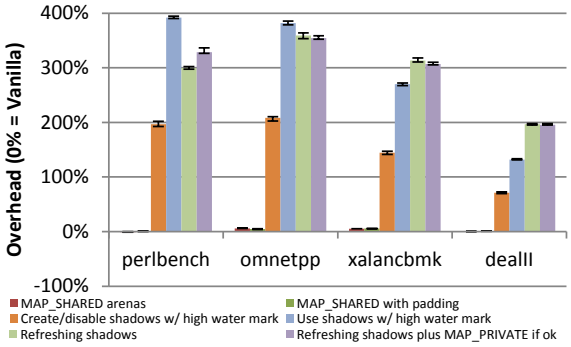

Figure 13: The 4 allocation-intensive benchmarks.

for `mprotect`'ed shadows corresponding to "freed" objects. The high water mark optimization, which permanently `munmap`s the shadows, allows Linux to reclaim the `vm_area_structs`, reducing the memory utilization significantly and enabling them to complete successfully. To separate out the cost of syscalls from TLB pressure, we backported the high water mark change to Create/disable shadows.

For all four benchmarks, `MAP_SHARED` and inline metadata costs (the first two columns) are insignificant compared to creating/disabling and using shadows. Refreshing shadows reduces overhead somewhat for `perlbench` and `omnetpp` but increases overhead for `xalancbmk` and `dealII`.

The `MAP_PRIVATE` optimization had a negligible effect, except for `perlbench`, which became 30 p.p. slower. This was initially surprising, since in all other cases, `MAP_PRIVATE` is faster than `MAP_SHARED`. However, recall that Oscar also had to change the `realloc` implementation. `perlbench` uses `realloc` heavily: 11 million calls, totaling 700GB of objects; this is 19x the `realloc`s of all other 18 benchmarks *combined* (by calls or GBs of objects). We confirmed that `realloc` caused the slowdown, by modifying Refreshing shadows to use the inefficient `realloc` but with `MAP_SHARED` always;

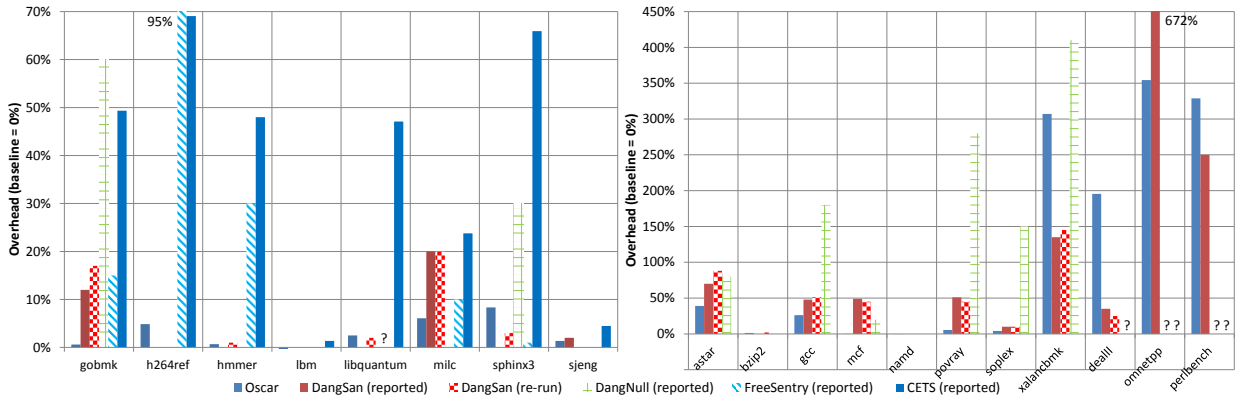

Figure 14: Runtime overhead of SPEC benchmarks. The graphs have different y-axes. Some overheads are based on results reported in the papers, not re-runs (see legend). '?' indicates that FreeSentry did not report results for `libquantum`, DangNull did not report results for `dealII`, `omnetpp`, or `perlbench`, and we could not re-run DangSan on `omnetpp` or `perlbench`. FreeSentry and CETS did not report results for any of the benchmarks in the right graph.

this was marginally slower than refreshing shadows and using `MAP_PRIVATE` where possible.

4.2 Runtime Overhead Comparison

Figure 14 (left) compares the runtime overhead of Oscar against DangSan, DangNull, FreeSentry, and CETS. Figure 14 (right) shows the remaining SPEC benchmarks, for which results were reported by DangSan and DangNull, but not FreeSentry or CETS.

A caveat is that CETS' reported overheads are based on providing temporal protection for both the stack and heap, which is more comprehensive than Oscar's heap-only protection. However, since CETS must, to a first approximation, fully instrument pointer arithmetic and dereferencing instructions even if only heap protection is desired, we expect that the overhead of heap-only CETS would still be substantially higher than Oscar.

All other comparisons (DangSan, DangNull, FreeSentry) are based on the appropriate reported overheads for heap-only temporal protection.

Comparison to DangSan. We re-ran the latest publicly available version of DangSan[3] on the same hardware as Oscar. DangSan re-run overheads were normalized to a re-run with their "baseline LTO" script. We were unable to re-run `perlbench` due to a segmentation fault, or `omnetpp` due to excessive memory consumption[4]. As seen in the graphs, our re-run results are very similar to DangSan's reported results; thus, unless otherwise stated, we will compare Oscar against the latter.

Across the complete set of C/C++ SPEC CPU2006 benchmarks, Oscar and DangSan have the same overall overhead, within rounding error (geometric means of 40% and 41%). However, for all four of the allocation-intensive benchmarks, as well as `astar` and `gcc`, the overheads of both Oscar and DangSan are well above the 10% overhead threshold [39], making it unlikely that either technique would be considered acceptable. If we exclude those six benchmarks, then Oscar has average overhead of 2.5% compared to 9.9% for DangSan. Alternatively, we can see that, for five benchmarks (`mcf`, `povray`, `soplex`, `gobmk`, `milc`), Oscar's overhead is 6% or less, whereas DangSan's is 10% or more. There are no benchmarks where DangSan has under 10% overhead but Oscar is 10% or more.[5]

Comparison to DangNull/FreeSentry. We emailed the first authors of DangNull and FreeSentry to ask for the source code used in their papers, but did not receive a response. Our comparisons are therefore based on the numbers reported in the papers rather than by re-running their code on our system. Nonetheless, the differences are generally large enough to show trends. In many cases, Oscar has almost zero overhead, implying there are few `mallocs`/`frees` (the source of Oscar's overhead); we expect the negligible overhead generalizes to any system. Oscar does not instrument the application's pointer arithmetic/dereferencing, which makes its overhead fairly insensitive to compiler optimizations. We also note that DangSan – which we were able to re-run and compare against Oscar – *theoretically* should have better performance than DangNull[6].

[3] March 19, 2017, `https://github.com/vusec/dangsan/commit/78006af30db70e42df25b7d44352ec717f6b0802`

[4] We estimate that it would require over 20GB of memory, taking into account the baseline memory usage on our machine and DangSan's reported overhead for `omnetpp`.

[5] Of course, there is a wide continuum of "under 10%", and those smaller differences may matter.

[6] However, DangSan's empirical comparisons to DangNull and FreeSentry were also based on reported numbers rather than re-runs.

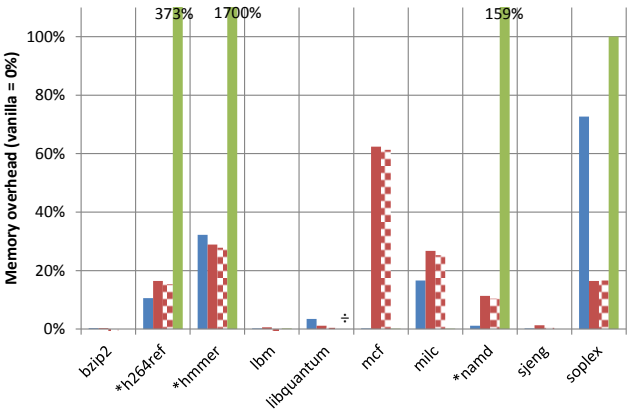

Figure 15: Memory overhead on CPU2006. DangNull reported a baseline of 0MB for libquantum, so an overhead ratio is not calculable.

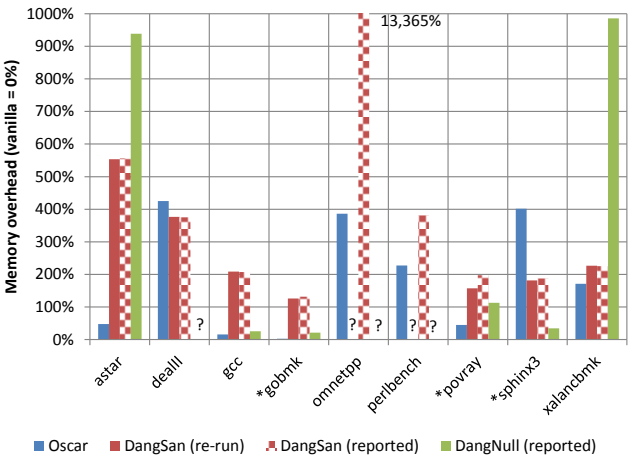

Figure 16: Memory overhead on CPU2006 (continued). '?' indicates that DangNull did not report memory usage for `dealII`, `omnetpp`, or `perlbench`, and we could not re-run DangSan on the latter two.

Oscar's performance is excellent compared to FreeSentry and DangNull, even though DangNull provides less comprehensive protection: DangNull only protects pointers to heap objects *if* the pointer is itself stored on the heap. Figure 14 (left) compares all SPEC CPU2006 benchmarks for which DangNull and FreeSentry both provide data. FreeSentry has higher overhead for several benchmarks (`milc`, `gobmk`, `hmmer`, `h264ref`) – especially higher for the latter three. FreeSentry is faster on the remaining three benchmarks, but in all those cases except for `sphinx3`, our overhead is negligible anyway. DangNull has much higher overhead than Oscar for `gobmk` and `sphinx3`. For other benchmarks, DangNull often gets zero overhead, though it is not much lower than Oscar's, and comes with the caveat of their weaker protection.

Our comparisons are based on our overall "best" scheme with all three optimizations. For some benchmarks, using just the high water mark optimization and not the other two optimizations would have performed better. Even the basic shadow pages scheme without optimizations would often beat DangNull/FreeSentry.

Figure 14 (right) shows additional SPEC CPU2006 benchmarks for which DangNull reported their overhead but FreeSentry did not. For the two benchmarks where DangNull has zero overhead (`bzip2`, `namd`), Oscar's are also close to zero. For the other six benchmarks, Oscar's overhead is markedly lower. Two highlights are `soplex` and `povray`, where DangNull's overhead is 150%/280%, while Oscar's is under 6%.

When considering only the subset of CPU2006 benchmarks that DangNull reports results for (i.e., excluding `dealII`, `omnetpp` and `perlbench`), Oscar has a geometric mean runtime overhead of 15.4% compared to 49% for DangNull. For FreeSentry's subset of reported benchmarks, Oscar has just 2.8% overhead compared to 18% for FreeSentry.

Comparison to CETS. We compare Oscar to the temporal-only mode of SoftBoundCETS [32] (which we will also call "CETS" for brevity), since that has lower overhead and a more comprehensive dataset than the original CETS paper.

The latest publicly available version of SoftBoundCETS for LLVM 3.4[7] implements both temporal and spatial memory safety. We received some brief advice from the author of SoftBoundCETS on how to modify it to run in temporal-only mode, but we were unable to get it to work beyond simple test programs. Thus, our comparisons rely on their reported numbers rather than a re-run.

We have omitted the `bzip2` and `mcf` benchmarks, as CETS' bzip2 is from the CPU2000 suite [29] and we suspect their mcf is as well.[8] SPEC specifically cautions that, due to differences in the benchmark workload and/or source, the results on CPU2000 vs. CPU2006 might not be comparable [5].

Figure 14 (left) shows the overhead of CETS vs. our overall best scheme. We are faster than CETS for all benchmarks, often by a significant margin. For example, CETS has >48% overhead on `gobmk` and `hmmer`, compared to less than 1% for Oscar. The geometric mean across CETS' subset of CPU2006 benchmarks is 2.8% for Oscar compared to 36% for CETS.

[7] September 19, 2014, `https://github.com/santoshn/softboundcets-34/commit/9a9c09f04e16f2d1ef3a906fd138a7b89df44996`

[8] In any case, since CETS has 23% and 114% overhead on `bzip2` and `mcf` respectively – compared to less than 1.5% on each for Oscar – including them in the comparison would not be favorable to CETS.

4.3 Memory Overhead Comparison

Figures 15 and 16 show the memory overhead of Oscar, DangSan (re-run and reported), and DangNull (reported only). We did not find any reported data for FreeSentry, CETS or SoftBoundCETS temporal-only. The graphs have different y-axes to highlight differences in overheads in the lower-overhead benchmarks of Figure 15.

We calculated the memory overhead based on the combined maximum resident set size (RSS)[9], size of the page tables[10], and approximate size of the `vm_area_structs`[11]. Our polling approach introduces some minor inaccuracies with regard to obtaining the maxima and baseline values. For DangSan, which does not greatly increase the number of page table entries or `vm_area_structs`, this is very similar to their maximum resident set size metric. It is unclear what memory consumption metric DangNull used, so some care should be taken when interpreting their overheads.

The RSS values reported in `/proc/pid/status` are misleading for Oscar because it double-counts every shadow page, even though many of them are aliased to the same canonical. We know, however, that the physical memory usage of Oscar – and therefore the resident set size when avoiding double-counting – is essentially the same as the MAP_SHARED with padding scheme (from Section 3.1). We therefore calculated the maximum RSS for that scheme, but measured the size of the page tables and `vm_area_structs` for the full version of Oscar.

For the complete suite of CPU2006 benchmarks, Oscar has 61.5% memory overhead, far lower than DangSan's 140%. Even if we omit DangSan's pathological case of `omnetpp` (reported overhead of over 13,000%), Oscar is still far more memory-efficient with 52% overhead vs. 90% for DangSan. The only benchmarks on which Oscar performs substantially worse than DangSan are `sphinx3` and `soplex`. `sphinx3` with Oscar has a maximum RSS of ≈50MB (compared to a baseline of ≈45MB), maximum page tables size of ≈130MB, and maximum `vm_area_structs` of ≈45MB. In Section 8, we propose methods to reduce the memory overhead by garbage collecting old page table entries (which would benefit `sphinx3`), and sharing inline metadata (which benefits would `soplex` with its many small allocations).

DangNull has roughly 127% memory overhead, but, as also noted by the DangSan authors, DangNull did not report data for many of the memory-intensive benchmarks. If we use the same subset of SPEC benchmarks that DangNull reported, then Oscar has only 36% memory overhead (vs. ≈75% for DangSan).

[9] `VmHWM` (peak RSS) in `/proc/pid/status`

[10] `VmPTE` and `VmPMD` in `/proc/pid/status`

[11] We counted the number of mappings in `/proc/pid/maps` and multiplied by `sizeof(vm_area_struct)`.

5 Extending Oscar for Server Applications

When applying Oscar to server applications – which are generally more complex than the SPEC CPU benchmarks – we encountered two major issues that resulted in incompatibility and incomplete protection: forking and custom memory allocators. Additionally, we modified Oscar to be thread-safe when allocating shadows.

5.1 Supporting shadows + fork()

Using `MAP_SHARED` for all allocations is problematic for programs that `fork`, as it changes the semantics of memory: the parent and child's memory will be shared, so any post-`fork` writes to pre-`fork` heap objects will unexpectedly be visible to both the parent and child. In fact, we discovered that most programs that `fork` and use `glibc`'s `malloc` will crash when using `MAP_SHARED`. Surprisingly, they may crash even if neither the parent nor child read or write to the objects post-`fork`.[12]

Oscar solves this problem by wrapping `fork` and emulating the memory semantics the program is expecting. After `fork`, in the child, we make a copy of all heap objects, unmap their virtual addresses from the shared physical page frames, remap the same virtual addresses to new (private) physical page frames, and repopulate the new physical page frames with our copy of the heap objects. The net effect is that the shadow and canonical virtual addresses have not changed – which means old pointers (in the application, and in the allocator's free lists) still work – but the underlying physical page frames in the child are now separated from the parent.

Method. Oscar instruments `malloc` and `free` to keep a record of all live objects in the heap and their shadow addresses. Note that with a loadable kernel module, Oscar could avoid recording the shadow addresses of live objects and instead find them from the page table entries or `vm_area_structs`.

Then, Oscar wraps `fork` to do the following:

1. call the vanilla `fork()`. After this, the child address space is correct, except that the `malloc`'d memory regions are aliased with the parent's physical page frames.

2. in the child process:

   (a) for each `canonical_page` in the heap:

[12] `glibc`'s `malloc` stores the main heap state in a static variable (not shared between parent and child), but also partly through inline metadata of heap objects (shared); thus, when the parent or child allocates memory post-`fork`, the heap state can become inconsistent or corrupted. A program that simply `malloc()s` 64 bytes of memory, `fork()s`, and then allocates another 64 bytes of memory in the child, is sufficient to cause an assertion failure.

i. allocate a new page at any unused address `t` using `mmap(MAP_SHARED | MAP_ANONYMOUS)`

ii. copy `canonical_page` to `t`

iii. call `mremap(old_address=t, new_address=canonical_page)`. Note that `mremap` automatically removes the previous mapping at `canonical_page`.

(b) for each live object: use `mremap` to recreate a shadow at the same virtual address as before (using the child's new physical page frames).

Compared to the naïve algorithm, the use of `mremap` halves the number of memory copy operations.

We can further reduce the number of system calls by observing that the temporary pages `t` can be placed at virtual addresses of our choice. In particular, we can place all the temporary pages in one contiguous block, which lets us allocate them all using just one `mmap` command.

The parent process must sleep until the child has copied the canonical pages, but it does not need to wait while the child patches up the child's shadows. Oscar blocks signals for the duration of the `fork()` wrapper.

This algorithm suffices for programs that have only one thread running when the program forks. This covers most reasonable use cases; it is considered poor practice to have multiple threads running at the time of `fork` [6]. For example, `apache`'s event multi-processing module forks multiple children, which each then create multiple threads. To cover the remaining, less common case of programs that arbitrarily mix threads and `fork`, Oscar could "stop the world" as in garbage collection, or LeakSanitizer (a memory leak detector) [1].

Our algorithm could readily be modified to be "copy-on-write" for efficiency. Additionally, batching the remappings of each page might improve performance; since the intended mappings are known in advance, we could avoid the misprediction issue that plagued regular batch mapping. With kernel support we could solve this problem more efficiently, but our focus is on solutions that can be deployed on existing platforms.

Results. We implemented the basic algorithm in Oscar. In cursory testing, `apache`, `nginx`, and `openssh` run with Oscar's `fork` fix, but fail without. These applications allocate only a small number of objects pre-`fork`, so Oscar's `fork` wrapper does not add much overhead (tens or hundreds of milliseconds).

5.2 Custom Memory Allocators

The overheads reported for SPEC CPU are based on instrumenting the standard `malloc`/`free` only, providing a level of protection similar to prior work. However, a few of the SPEC benchmarks [19] implement their own custom memory allocator (CMAs). Since standard schemes for temporal memory safety require instrumenting memory allocation and deallocation functions, without special provisions none of them – including Oscar – will protect objects allocated via arbitrary CMAs.

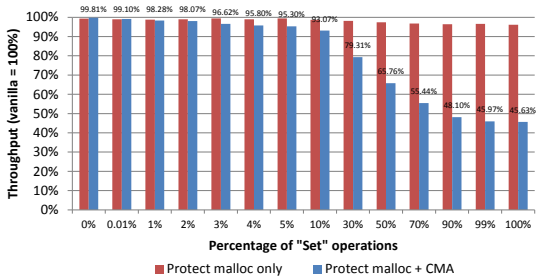

Figure 17: Throughput of Oscar on memcached.

We found that CMAs seem to be even more common in server programs, such as `apache`, `nginx`, and `proftpd`. Prior work typically ignores the issue of CMAs. We solve this by manually identifying CMAs and wrapping them with Oscar as well. CMA identification could also be done automatically [18].

If we do not wrap a CMA with Oscar, any objects allocated with the CMA would obviously not be resistant to use-after-free. However, there are no other ill effects; it would not result in any false positives for any objects, nor would it result in false negatives for the non-CMA objects.

5.3 Case Study: malloc-like custom memory allocator in memcached

`memcached` is a memory object caching system that exports a get/set interface to a key-value store. We compiled `memcached 1.4.25` (and its prerequisite, `libevent`) and benchmarked performance using `memaslap`.

When we wrapped only `glibc`'s `malloc`, the overhead was negligible: throughput was reduced by 0–3%, depending on the percentage of set operations (Figure 17). However, this is misleadingly low, as it fails to provide temporal memory safety for objects allocated by the CMA. Therefore, we applied Oscar to wrap the CMA, in the same way we wrapped glibc's `malloc`/`free`.

Method. To support wrapping the CMA, we had to ensure that Oscar `malloc` always used `MAP_SHARED` even for large objects. This is because the allocation may be used by the CMA to "host" a number of shadows. Additionally, we partitioned the address space to use separate high-water marks for the `malloc` wrapper and CMA wrapper.

We identified that allocations and deallocations via

memcached's slab allocator are all made through the `do_item_alloc` and `item_free` functions. Thus, it is sufficient to add shadow creation/deletion to those functions.

For ease of engineering, we made minor changes directly to the slab allocator, similar to those we applied to `glibc`'s `malloc`: inserting a canonical address field in the memcached item struct, and modifying the allocation/deallocation functions. In principle, we only need to override the CMA allocate/deallocate symbols, without needing to recompile the main application.

In this paper, the per-object metadata (e.g., the canonical address) is stored inline. If Oscar switched to a disjoint metadata store (e.g., a hashtable), it would be easy to extend Oscar to protect any custom memory allocators (not just CMAs with `malloc`-like interfaces) that are identified: as with `glibc`'s `malloc`, the allocator function simply needs to be wrapped to return a new shadow, and the deallocator function wrapped to destroy the shadow. This would be a better long-term approach than individually dealing with each CMA that is encountered.

Results. When set operations are 3% of the total operations (a typical workload [12]), the performance overhead is roughly 4%. The overhead is higher for set operations because these require allocations (via the CMA), which involves creating shadows. Get operations have almost no overhead because they do not perform memory allocation or deallocation and consequently do not require any system calls.[13] Unlike SPEC CPU, which is single-threaded, we ran `memcached` with 12 threads. This shows that Oscar's overhead is low even for multi-threaded applications, despite our naïve use of a mutex to synchronize part of Oscar's internal state (namely, the high-water mark; see Section 8).

5.4 Special case: Region-based allocators

We have found several server programs that use region-based custom memory allocators [14]. Region-based allocators are particularly favorable for page-permissions-based schemes such as Oscar.

Typically, region-based allocators obtain a large block of memory from `malloc`, which they carve off into objects for their allocations. The distinguishing feature is that only the entire region can be freed, but not individual objects.

Region-based allocators by themselves are not resistant to use-after-free, since the blocks from `malloc` may be reused, but they provide temporal memory safety when the underlying `malloc/free` is protected by a lock-and-key scheme. Thus, there is no need to explicitly identify region-based CMAs; merely wrapping glibc's `malloc/free` with Oscar suffices to provide temporal memory safety for such programs i.e., Oscar would provide full use-after-free protection for a region-based allocator, without the need for any custom modifications.

Oscar's performance is especially good for programs that use region-based allocators: since there are few `malloc()`s or `free()`s to instrument, and correspondingly low memory or TLB pressure, Oscar imposes negligible overhead. Other classes of lock-and-key schemes also provide full protection to programs with region-based allocators, but they often have high overhead, since they must instrument all pointer arithmetic operations (and possibly pointer dereferences).

6 Discussion

Our results show that shadow-page-based schemes with our optimizations have low overhead on many benchmarks. From Table 1, we argue that changing the lock is theoretically easier than revoking all the keys, and implicit lock-and-key is better than explicit. Our experimental results confirm that prediction: Oscar's runtime overhead is lower than CETS, DangNull, and FreeSentry overall and on most benchmarks, and comparable to DangSan (but with lower memory overhead for Oscar), even though they all need source code while Oscar does not.

6.1 Virtual Address Space Considered Hard to Fill

A concern might be that Oscar would exhaust the $2^{47}B$ =128TB user-space virtual address space, necessitating reuse of addresses belonging to freed pages. This is unlikely in common scenarios. Based on extrapolating the CPU2006 benchmarks, it would take several *days* of continuous execution even for allocation-intensive programs. For example, with `perlbench`, which allocates 361 million objects (≈1.4TB of shadow virtual pages; >99% of objects fit in one page) over 25 minutes, it would take 1.6 days (albeit less on newer, faster hardware) to allocate 128TB. `dealII`, `omnetpp` and `xalancbmk` would take over 2.5 days each, `gcc` would take 5 days, and all other CPU2006 benchmarks would take at least 2 *weeks*. We expect that most programs would have significantly shorter lifetimes, and therefore would never exhaust the virtual address space. It is more likely that they would first encounter problems with the unreclaimed page-table memory (see Section 8). Nonetheless, it is possible to ensure safe reuse of virtual address space, by applying a conservative garbage collector to old shadow addresses (note that this does not

[13] Technicality: `memcached` lazily expires entries, checking the timestamp only during the get operation. Thus, the overhead of destroying shadows may be attributed to get operations.

affect physical memory, which is already reused with new shadow addresses); this was proposed (but not implemented) by Dhurjati and Adve.

Recently, Intel has proposed 5-level paging, allowing a 57-bit virtual address space [20]; implementation of Linux support is already underway [37]. This 512-fold increase would make virtual address space exhaustion take *years* for every CPU2006 benchmark.

6.2 Hardware Extensions

Due to the high overhead of software-based temporal memory safety for C, some have proposed hardware extensions (e.g., Watchdog [30]). Oscar is fast because it *already* utilizes hardware – hardware which is present in many generations of x86 CPUs: the memory management unit, which checks page table entries. We believe that, with incremental improvements, shadow-page-based schemes will be fast enough for widespread use, without the need for special hardware extensions. For example, Intel's Broadwell CPUs have a larger TLB and also a second TLB page miss handler [7], which are designed to improve performance for general workloads, but would be particularly useful in relieving Oscar's TLB pressure. Intel has also proposed finer grained memory protection [35]; if future CPUs support read+write protection on subpage regions, Oscar could be adapted to one-object-per-*subpage*, which would reduce the number of shadows (and thereby TLB pressure).

6.3 Compatibility

Barring virtual address space exhaustion (discussed in Section 6.1), Oscar will crash a program if and only if the program *dereferences* a pointer after its object has been freed. It does not interfere with other uses of pointers. Unlike other lock-and-key schemes, page-permissions-based schemes do not need to instrument pointer arithmetic or dereferencing (Table 1).

Accordingly, Oscar correctly handles many corner cases that other schemes cannot handle. For example, DangNull/FreeSentry do not work correctly with encrypted pointers (e.g., PointGuard [21]) or with typecasting from non-pointer types. CETS has false positives when casting from a non-pointer to pointer, as it will initialize the key and lock address to invalid values.

Additionally, DangNull does not allow pointer arithmetic on freed pointers. For example, suppose we allocate a string `p` on the heap, search for a character, then free the string:

```
char* p = strdup("Oscar"); // Memory from malloc
char* q = strchr(p, 'a');  // Find the first 'a'
free(p);
```

Computing the index of "a" (`q - p == 3`) fails with DangNull, since `p` and `q` were nullified. It does work with DangSan and FreeSentry (since they only change the top bits) and with Oscar.

DangSan, DangNull and FreeSentry only track the location of pointers when they are stored in memory, but not registers. This can lead to false negatives: DangSan notes that this may happen with pointers spilled from registers onto the stack during function prologues, as well as race conditions where a pointer may be stored into a register by one thread while another thread frees that object. DangSan considers both issues to be infeasible to solve (for performance reasons, and also the possibility of false positives when inspecting the stack).

7 Related Work

7.1 Dhurjati and Adve (2006)

Our work is inspired by the original page-permission with shadows scheme by Dhurjati and Adve [23]. Unlike Dhurjati and Adve's automatic pool allocation, Oscar can unmap shadows as soon as an object is freed, and does not require source code. Oscar also addresses compatibility with `fork`, which appears to be a previously unknown limitation of Dhurjati and Adve's scheme[14]. They considered programs that `fork` to be advantageous, since virtual address space wastage in one child will not affect the address space of other children. Unfortunately, writes to old (pre-`fork`) heap objects will be propagated between parent and children (see Section 5.1), resulting in memory corruption.

While Dhurjati and Adve did measure the runtime of their particular scheme, their measurements do not let us break down how much each aspect of their scheme contributes to runtime overhead. First, their scheme relies upon static analysis (*Automatic Pool Allocation*: "PA"), and they did not measure the cost of shadow pages without PA. We cannot simply obtain "cost of syscalls" via "(PA + dummy syscalls) − PA", since pool allocation affects the cost of syscalls and cache pressure. Second, they did not measure the cost of each of the four factors we identified. For instance, they did not measure the individual cost of inline metadata or changing the memory allocation method; instead, they are lumped in with the cost of dummy syscalls. This makes it hard to predict the overhead of other variant schemes, e.g., using one object per physical page frame. Finally, they used a custom benchmark and Olden [34], which make it harder to compare their results to other schemes that are benchmarked with SPEC CPU; and many of their benchmark

[14] We inspected their source `http://safecode.cs.illinois.edu/downloads.html` and found that they used `MAP_SHARED` without a mechanism to deal with `fork`.

	One object per physical page frame	One object per shadow virtual page (core technique of Dhurjati & Adve [D&A])		
Physical memory overhead	**e.g., Electric Fence**	**Vanilla**	**Automatic pool allocation [D&A]**	**Our work**
User-space memory	0 – 4KB per object (page align)	✓ Low overhead (O(sizeof(void*)) per object)		
Page table entry for live objects	1 page table entry per object			
Page table entry for freed objs	<Depends on implementation>	1 PTE per object	1 PTE per object in live pools	0 PTEs*
VMA struct for live objects	1 VMA struct per object			
VMA struct for freed objects	<Depends on implementation>	1 VMA struct per object		✓ None
No application source needed	✓ Yes	✓ Yes	No; needs source + recompilation	✓ Yes
Compatible with fork()	✓ Yes	No; changes program semantics		✓ Mostly

Table 2: Comparison with Dhurjati and Adve. Green and a tick indicates an advantageous distinction. ∗ Oscar unmaps the shadows for freed objects, but Linux does not reclaim the PTE memory (see Section 8).

run-times are under five seconds, which means random error has a large impact. For these reasons, in this work we undertook a more systematic study of the sources of overhead in shadow-page-based temporal memory safety schemes.

To reduce their system's impact on page table utilization, Dhurjati and Adve employed static source-code analysis (Automatic Pool Allocation) – to separate objects into memory pools of different lifetimes, beyond which the pointers are guaranteed not to be dereferenced. Once the pool can be destroyed, they can remove (or reuse) page table entries (and associated `vm_area_structs`) of freed objects. Unfortunately, there may be a significant lag between when the object is freed, and when its containing pool is destroyed; in the worst case (e.g., objects reachable from a global pointer), a pool may last for the lifetime of the program. Besides being imprecise, inferring object lifetimes via static analysis also introduces a requirement to have application source code, making it difficult and error-prone to deploy. Oscar's optimizations do not require application source code or compiler changes.

We cannot directly compare Oscar's overhead to Dhurjati and Adve's full scheme with automatic pool allocation, since they did not report numbers for SPEC CPU.

Oscar usually keeps less state for freed objects: they retain a page table entry (and associated `vm_area_struct`) for each freed object in live pools – some of which may be long-lived – whereas Oscar `munmaps` the shadow as soon as the object is freed (Table 2). Dhurjati and Adve expressly target their scheme towards server programs – since those do few allocations or deallocations – yet they do not account for `fork` or custom memory allocators.

If we are not concerned about the disadvantages of automatic pool allocation, it too would benefit from our optimizations. For example, we have seen that using `MAP_PRIVATE` greatly reduces the overhead for `mcf` and `milc`, and we expect this benefit to carry over when combined with automatic pool allocation.

7.2 Other Deterministic Protection Schemes

The simplest protection is to never `free()` any memory regions. This is perfectly secure, does not require application source code (change the `free` function to be no-op), has excellent compatibility, and low run-time overhead. However, it also requires infinite memory, which is impractical.

With DangNull [27], when an object is freed, all pointers to the object are set to NULL. The converse policy – when all references to a region are NULL (or invalid), automatically `free` the region – is "garbage collection". In C/C++, there is ambiguity about what is a pointer, hence it is only possible to perform conservative garbage collection, where anything that might plausibly be a pointer is treated as a pointer, thus preventing `free()`'ing of the referent. This has the disadvantages of false positives and lower responsiveness.

The Rust compiler enforces that each object can only have one owner [4]; with our lock-and-key metaphor, this is equivalent to ensuring that each lock has only one key, which may be "borrowed" (ala Rust terminology) but not copied. This means that when a key is surrendered (pointer becomes out of scope), the corresponding lock-/object can be safely reused. It would be impractical to rewrite all legacy C/C++ software in Rust, let alone provide Rust's guarantees to binaries that are compiled from C/C++.

MemSafe [38] combines spatial and temporal memory checks: when an object is deallocated, the bounds are set to zero (a special check is required for sub-object temporal memory safety). MemSafe modifies the LLVM IR, and does not allow inline assembly or self-modifying code. Of the five SPEC 2006 benchmarks they used, their run-times appear to be from the 'test' dataset rather than the 'reference' dataset. For example, for `astar`, their base run-time is 0.00 seconds, whereas Oscar's is 408.9 seconds. Their non-zero run-time benchmarks have significant overhead – 183% for `bzip2`, 127% for `gobmk`, 124% for `hmmer`, and 120% for `sjeng` – though this in-

cludes spatial and stack temporal protection.

Dynamic instrumentation (e.g., Valgrind's memcheck [3]) is generally too slow other than for debugging.

Undangle [15] uses taint tracking to track pointer propagation. They do not provide SPEC results, but we expect it to be even slower than DangNull/FreeSentry, because Undangle determines how pointers are propagated by, in effect, interpreting each x86 instruction.

Safe dialects of C, such as CCured [33], generally require some source code changes, such as removing unsafe casts to pointers. CCured also changes the memory layout of pointers (plus-size pointers), making it difficult to interface with libraries that have not been recompiled with CCured.

7.3 Hardening

The premise of heap temporal memory safety schemes, such as Oscar, is that the attacker could otherwise repeatedly attempt to exploit a memory safety vulnerability, and has disabled or overcome any mitigations such ASLR (nonetheless, as noted earlier, Oscar is compatible with ASLR). Thus, Oscar provides deterministic protection against heap use-after-free (barring address space exhaustion/reuse, as discussed in Section 6.1).

However, due to the high overhead of prior temporal memory safety schemes, some papers trade off protection for speed.

Many papers, starting with DieHard [13], approximate the infinite heap (use a heap that is M times larger than normally needed) and randomize where objects are placed on the heap. This means even if an object is used after it is freed, there is a "low" probability that the memory region has been reallocated. Archipelago [28] extends DieHard but uses less physical memory, by compacting cold objects. Both can be attacked by making many large allocations to exhaust the M-approximate heap, forcing earlier reuse of freed objects.

AddressSanitizer [36] also uses a quarantine pool, though with a FIFO reuse order, among other techniques. PageHeap [2] places freed pages in a quarantine, with the read/write page permissions removed. Attempted reuse will be detected only if the page has not yet been reallocated, so it may miss some attacks. These defenses can also be defeated by exhausting the heap.

Microsoft's MemoryProtection consists of Delayed Free (similar to a quarantine) and Isolated Heap (which separates normal objects from "critical" objects) [8]. Both of these defenses can be bypassed [22].

Cling [11] only reuses memory among heap objects of the same type, so it ensures type-safe heap memory reuse but not full heap temporal memory safety.

7.4 Limiting the Damage from Exploits

Rather than attempting to enforce memory safety entirely, which may be considered too expensive, some approaches have focused on containing the exploit.

Often the goal of exploiting a user-after-free vulnerability is to hijack the control flow, such as by modifying function pointers per our introductory example. One defense is control-flow integrity (CFI) [10], but recent work on "control-flow bending" [16] has shown that even the ideal CFI policy may admit attacks for some programs. Code pointer integrity (CPI) is essentially memory safety (spatial and temporal) applied only to code pointers [26]. Code pointer separation (CPS) is a weaker defense than CPI, but stronger than CFI. Both CPI and CPS require compiler support.

CFI, CPS and CPI do not help against non-control data attacks, such as reading a session key or changing an 'isAdmin' variable [17]; recently, "data-oriented programming" has been shown to be Turing-complete [24].

8 Limitations and Future Work

Oscar is only a proof-of-concept for measuring the overhead on benchmarks, and is not ready for production, primarily due to the following two limitations.

Reclaiming page-table memory takes some engineering, such as using `pte_free()`. Alternatively, the Linux source mentions they "Should really implement gc for free page table pages. This could be done with a reference count in struct page."[15] Not all page-tables can be reclaimed, as some page-tables may contain entries for a few long-lived objects, but the fact that most objects are short-lived (the "generational hypothesis" behind garbage collection) suggests that reclamation may be possible for many page-tables. Note that the memory overhead comparison in Section 4.3 already counts the size of paging structures against Oscar, yet Oscar still has lower overall overhead despite not cleaning up the paging structures at all.

We did not encounter any issues with users' `mmap` requests overlapping Oscar's region of shadow addresses (or vice-versa), but it would be safer to deterministically enforce this by intercepting the users' `mmap` calls.

Currently, all threads share the same high-water mark for placing new shadows, and this high-water mark is protected with a global mutex. A better approach would be to dynamically partition the address space between threads/arenas; for example, when a new allocator arena is created, it could split half the address space from the arena that has the current largest share of the address space. Each arena could therefore have its own

[15] http://lxr.free-electrons.com/source/arch/x86/include/asm/pgalloc.h

high-water mark, and allocations could be made independently of other arenas. This could lower the overhead of the `memcached` benchmarks, but not the SPEC CPU benchmarks (which are all single-threaded).

Our techniques could be applied to other popular memory allocators (e.g., `tcmalloc`), or more generally, any custom memory allocator. The overheads reported for SPEC CPU are based on instrumenting the standard `malloc`/`free` only, providing a level of protection similar to prior work. Wrapping CMA's provides more comprehensive protection, though the overheads would be higher for a few benchmarks, as discussed in Section 5.2.

If we are willing to modify `internal_malloc`, Oscar can be selective in how to refresh (or batch-create) shadows. For example, objects that are small enough (among other conditions) to fit in `internal_malloc`'s "small bins" are reused in a first-in-first-out order, which means that a speculatively created shadow is likely to be used eventually. Other bins are last-in-first-out or even best-fit, which makes their future use less predictable. This optimization may particularly benefit `xalancbmk` and `dealII`, for which the ordinary refresh shadow approach was a net loss.

We could take advantage of the short-lived nature of most objects to experiment with placing multiple objects per shadow; fewer shadows means lower runtime and memory overhead. To further reduce memory overhead, we could change `internal_malloc` to place the canonical address field at the start of each page, rather than the start of each object. All objects on the page would then share the canonical address field, which could drastically reduce the memory overhead for programs with many small allocations (e.g., `soplex`).

9 Conclusion

Efficient, backwards compatible, temporal memory safety for C programs is a challenging, unsolved problem. By viewing many of the existing schemes as lock-and-key, we showed that page-permissions-based protection schemes were the most elegant and theoretically promising. We built upon Dhurjati and Adve's core idea of one shadow per object. That idea is unworkable by itself due to accumulation of `vm_area_structs` for freed objects and incompatibility with programs that `fork()`. Dhurjati and Adve's combination of static analysis partially solves the first issue but not the second, and comes with the cost of requiring source-code analysis. Our system Oscar addresses both issues and introduces new optimizations, all without needing source code, providing low overheads for many benchmarks and simpler deployment. Oscar thereby brings page-permissions-based protection schemes to the forefront of practical solutions for temporal memory safety.

10 Acknowledgements

This work was supported by the AFOSR under MURI award FA9550-12-1-0040, Intel through the ISTC for Secure Computing, and the Hewlett Foundation through the Center for Long-Term Cybersecurity.

We thank Nicholas Carlini, David Fifield, Úlfar Erlingsson, and the anonymous reviewers for helpful comments and suggestions.

References

[1] AddressSanitizerLeakSanitizer. `https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer`.

[2] How to use Pageheap.exe in Windows XP, Windows 2000, and Windows Server 2003. `https://support.microsoft.com/en-us/kb/286470`.

[3] Memcheck: a memory error detector. `http://valgrind.org/docs/manual/mc-manual.html`.

[4] Ownership and moves. `https://rustbyexample.com/scope/move.html`.

[5] Readme 1st CPU2006. `https://www.spec.org/cpu2006/Docs/readme1st.html#Q21`.

[6] Threads and fork(): think twice before mixing them. `https://www.linuxprogrammingblog.com/threads-and-fork-think-twice-before-using-them`, June 2009.

[7] Advancing Moore's Law in 2014! `http://www.intel.com/content/dam/www/public/us/en/documents/presentation/advancing-moores-law-in-2014-presentation.pdf`, August 2014.

[8] Efficacy of MemoryProtection against use-after-free vulnerabilities. `http://community.hpe.com/t5/Security-Research/Efficacy-of-MemoryProtection-against-use-after-free/ba-p/6556134#.VsFYB8v8vCK`, July 2014.

[9] Electric Fence. `http://elinux.org/index.php?title=Electric_Fence&oldid=369914`, January 2015.

[10] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. Control-flow integrity principles, implementations, and applications. *TISSEC* (2009).

[11] AKRITIDIS, P. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *USENIX Security* (2010), pp. 177–192.

[12] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review* (2012), vol. 40, ACM, pp. 53–64.

[13] BERGER, E. D., AND ZORN, B. G. DieHard: probabilistic memory safety for unsafe languages. *ACM SIGPLAN Notices 41*, 6 (2006), 158–168.

[14] BERGER, E. D., ZORN, B. G., AND MCKINLEY, K. S. Reconsidering custom memory allocation. *ACM SIGPLAN Notices 48*, 4S (2013), 46–57.

[15] CABALLERO, J., GRIECO, G., MARRON, M., AND NAPPA, A. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *International Symposium on Software Testing and Analysis* (2012), ACM, pp. 133–143.

[16] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security* (2015), pp. 161–176.

[17] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security* (2005), vol. 5.

[18] CHEN, X., SLOWINSKA, A., AND BOS, H. Who allocated my memory? Detecting custom memory allocators in C binaries. In *WCRE* (2013), pp. 22–31.

[19] CHEN, X., SLOWINSKA, A., AND BOS, H. On the detection of custom memory allocators in C binaries. *Empirical Software Engineering* (2015), 1–25.

[20] CORPORATION, I. 5-Level Paging and 5-Level EPT. `https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf`, May 2017.

[21] COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. PointGuard: protecting pointers from buffer overflow vulnerabilities. In *USENIX Security* (2003), vol. 12, pp. 91–104.

[22] DEMOTT, J. UaF: Mitigation and Bypass. `https://bromiumlabs.files.wordpress.com/2015/01/demott_uaf_migitation_and_bypass2.pdf`, January 2015.

[23] DHURJATI, D., AND ADVE, V. Efficiently detecting all dangling pointer uses in production servers. In *Dependable Systems and Networks* (2006), IEEE, pp. 269–280.

[24] HU, H., SHINDE, S., ADRIAN, S., CHUA, Z. L., SAXENA, P., AND LIANG, Z. Data-Oriented Programming: On the Expressive of Non-Control Data Attacks. In *IEEE S&P* (2016).

[25] JIM, T., MORRISETT, J. G., GROSSMAN, D., HICKS, M. W., CHENEY, J., AND WANG, Y. Cyclone: A Safe Dialect of C. In *USENIX ATC* (2002), pp. 275–288.

[26] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *OSDI* (2014), pp. 147–163.

[27] LEE, B., SONG, C., JANG, Y., WANG, T., KIM, T., LU, L., AND LEE, W. Preventing Use-after-free with Dangling Pointers Nullification. In *NDSS* (2015).

[28] LVIN, V. B., NOVARK, G., BERGER, E. D., AND ZORN, B. G. Archipelago: trading address space for reliability and security. *ACM SIGOPS Operating Systems Review 42*, 2 (2008), 115–124.

[29] NAGARAKATTE, S. personal communication, June 2017.

[30] NAGARAKATTE, S., MARTIN, M. M., AND ZDANCEWIC, S. Watchdog: Hardware for safe and secure manual memory management and full memory safety. *ACM SIGARCH Computer Architecture News 40*, 3 (2012), 189–200.

[31] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. CETS: compiler enforced temporal safety for C. *ACM Sigplan Notices 45*, 8 (2010), 31–40.

[32] NAGARAKATTE, S. G. *Practical low-overhead enforcement of memory safety for C programs*. University of Pennsylvania, 2012. Doctoral dissertation.

[33] NECULA, G. C., MCPEAK, S., AND WEIMER, W. CCured: Type-safe retrofitting of legacy code. *ACM SIGPLAN Notices 37*, 1 (2002), 128–139.

[34] ROGERS, A., CARLISLE, M. C., REPPY, J. H., AND HENDREN, L. J. Supporting dynamic data structures on distributed-memory machines. *TOPLAS 17*, 2 (1995), 233–263.

[35] SAHITA, R. L., SHANBHOGUE, V., NEIGER, G., EDWARDS, J., OUZIEL, I., HUNTLEY, B. E., SHWARTSMAN, S., DURHAM, D. M., ANDERSON, A. V., LEMAY, M., ET AL. Method and apparatus for fine grain memory protection, Dec. 31 2015. US Patent 20,150,378,633.

[36] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. AddressSanitizer: A fast address sanity checker. In *USENIX ATC* (2012), pp. 309–318.

[37] SHUTEMOV, K. A. [RFC, PATCHv1 00/28] 5-level paging. `http://lkml.iu.edu/hypermail/linux/kernel/1612.1/00383.html`, Dec 2016.

[38] SIMPSON, M. S., AND BARUA, R. K. MemSafe: ensuring the spatial and temporal memory safety of C at runtime. *Software: Practice and Experience 43*, 1 (2013), 93–128.

[39] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal war in memory. In *IEEE S&P* (2013), IEEE, pp. 48–62.

[40] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, Ú., LOZANO, L., AND PIKE, G. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security* (2014).

[41] VAN DER KOUWE, E., NIGADE, V., AND GIUFFRIDA, C. DangSan: Scalable Use-after-free Detection. In *EuroSys* (2017), pp. 405–419.

[42] YOUNAN, Y. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS* (2015).