Cling: A Memory Allocator to Mitigate Dangling Pointers

Periklis Akritidis

Niometrics, Singapore, and
University of Cambridge, UK

Abstract

Use-after-free vulnerabilities exploiting so-called dangling pointers to deallocated objects are just as dangerous as buffer overflows: they may enable arbitrary code execution. Unfortunately, state-of-the-art defenses against use-after-free vulnerabilities require compiler support, pervasive source code modifications, or incur high performance overheads. This paper presents and evaluates Cling, a memory allocator designed to thwart these attacks at runtime. Cling utilizes more address space, a plentiful resource on modern machines, to prevent type-unsafe address space reuse among objects of different types. It infers type information about allocated objects at runtime by inspecting the call stack of memory allocation routines. Cling disrupts a large class of attacks against use-after-free vulnerabilities, notably including those hijacking the C++ virtual function dispatch mechanism, with low CPU and physical memory overhead even for allocation intensive applications.

1 Introduction

Dangling pointers are pointers left pointing to deallocated memory after the object they used to point to has been freed. Attackers may use appropriately crafted inputs to manipulate programs containing use-after-free vulnerabilities [18] into accessing memory through dangling pointers. When accessing memory through a dangling pointer, the compromised program assumes it operates on an object of the type formerly occupying the memory, but will actually operate on whatever data happens to be occupying the memory at that time.

The potential security impact of these, so called, temporal memory safety violations is just as serious as that of the better known spatial memory safety violations, such as buffer overflows. In practice, however, use-after-free vulnerabilities were often dismissed as mere denial-of-service threats, because successful exploitation for arbitrary code execution requires sophisticated control over the layout of heap memory. In one well publicized case, flaw CVE-2005-4360 [17] in Microsoft IIS remained unpatched for almost two years after being discovered and classified as low-risk in December 2005.

Use-after-free vulnerabilities, however, are receiving increasing attention by security researchers and attackers alike. Researchers have been demonstrating exploitation techniques, such as heap spraying and heap feng shui [21, 1], that achieve the control over heap layout necessary for reliable attacks, and several use-after-free vulnerabilities have been recently discovered and fixed by security researchers and software vendors. By now far from a theoretical risk, use-after-free vulnerabilities have been used against Microsoft IE in the wild, such as CVE-2008-4844, and more recently CVE-2010-0249 in the well publicized attack on Google's corporate network.

Such attacks exploiting use-after-free vulnerabilities may become more widespread. Dangling pointers likely abound in programs using manual memory management, because consistent manual memory management across large programs is notoriously error prone. Some dangling pointer bugs cause crashes and can be discovered during early testing, but others may go unnoticed because the dangling pointer is either not created or not dereferenced in typical execution scenarios, or it is dereferenced before the pointed-to memory has been reused for other objects. Nevertheless, attackers can still trigger unsafe dangling pointer dereferences by using appropriate inputs to cause a particular sequence of allocation and deallocation requests.

Unlike omitted bounds checks that in many cases are easy to spot through local code inspection, use-after-free bugs are hard to find through code review, because they require reasoning about the state of memory accessed by a pointer. This state depends on previously executed code, potentially in a different network request. For the same reasons, use-after-free bugs are also hard to find through automated code analysis. Moreover, the combination of manual memory management and object oriented programming in C++ provides fertile ground for attacks, because, as we will explain in Section 2.1, the virtual function dispatch mechanism is an ideal target for dangling pointer attacks.

While other memory management related security problems, including invalid frees, double frees, and heap metadata overwrites, have been addressed efficiently and transparently to the programmer in state-of-the-art memory allocators, existing defenses against use-after-free vulnerabilities incur high overheads or require compiler support and pervasive source code modifications.

In this paper we describe and evaluate Cling, a memory allocator designed to harden programs against use-after-free vulnerabilities transparently and with low overhead. Cling constrains memory allocation to allow address space reuse only among objects of the same type. Allocation requests are inferred to be for objects of the same type by inspecting the allocation routine's call stack under the assumption that an allocation site (*i.e.* a call site of `malloc` or `new`) allocates objects of a single type or arrays of objects of a single type. Simple wrapper functions around memory allocation routines (for example, the typical `my_malloc` or `safe_malloc` wrappers checking the return value of `malloc` for `NULL`) can be detected at runtime and unwound to recover a meaningful allocation site. Constraining memory allocation this way thwarts most dangling pointer attacks —importantly— including those attacking the C++ virtual function dispatch mechanism, and has low CPU and memory overhead even for allocation intensive applications.

These benefits are achieved at the cost of using additional address space. Fortunately, sufficient amounts of address space are available in modern 64-bit machines, and Cling does not leak address space over time, because the number of memory allocation sites in a program is constant. Moreover, for machines with limited address space, a mechanism to recover address space is sketched in Section 3.6. Although we did not encounter a case where the address space of 32-bit machines was insufficient in practice, the margins are clearly narrow, and some applications are bound to exceed them. In the rest of this paper we assume a 64-bit address space—a reasonable requirement given the current state of technology.

The rest of the paper is organized as follows. Section 2 describes the mechanics of dangling pointer attacks and how type-safe memory reuse defeats the majority of attacks. Section 3 describes the design and implementation of Cling, our memory allocator that enforces type-safe address space reuse at runtime. Section 4 evaluates the performance of Cling on CPU bound benchmarks with many allocation requests, as well as the Firefox web browser (web browsers have been the main target of use-after-free attacks so far). Finally, we survey related work in Section 5 and conclude in Section 6.

2 Background

2.1 Dangling Pointer Attacks

Use-after-free errors are, so called, temporal memory safety violations, accessing memory that is no longer valid. They are duals of the better known spatial memory safety violations, such as buffer overflows, that access memory outside prescribed bounds. Temporal memory safety violations are just as dangerous as spatial memory safety violations. Both can be used to corrupt memory with unintended memory writes, or leak secrets through unintended memory reads.

When a program accesses memory through a dangling pointer during an attack, it may access the contents of some other object that happens to occupy the memory at the time. This new object may even contain data legitimately controlled by the attacker, *e.g.* content from a malicious web page. The attacker can exploit this to hijack critical fields in the old object by forcing the program to read attacker supplied values through the dangling pointer instead.

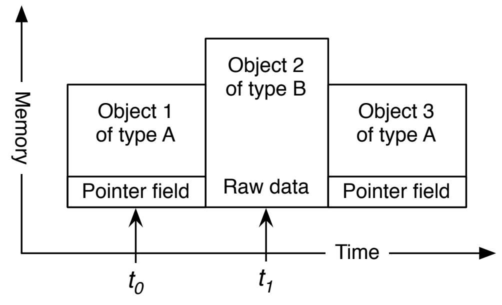

Figure 1: Unsafe memory reuse with dangling pointer.

For example, if a pointer that used to point to an object with a function pointer field (*e.g.* object 1 at time t_0 in Figure 1) is dereferenced to access the function pointer after the object has been freed, the value read for the function pointer will be whatever value happens to occupy the object's memory at the moment (*e.g.* raw data from object 2 at time t_1 in Figure 1). One way to exploit this is for the attacker to arrange his data to end up in the memory previously occupied by the object pointed by the dangling pointer and supply an appropriate value within his data to be read in place of the function pointer. By triggering the program to dereference the dangling pointer, the attacker data will be interpreted as a function pointer, diverting program control flow to the location dictated by the attacker, *e.g.* to shellcode (attacker code smuggled into the process as data).

Placing a buffer with attacker supplied data to the exact location pointed by a danging pointer is complicated by unpredictability in heap memory allocation. However, the technique of heap spraying can address this challenge with high probability of success by allocating large amounts of heap memory in the hope that some of it will end up at the right memory location. Alternatively, the attacker may let the program dereference a random function pointer, and similarly to uninitialized memory access exploits, use heap spraying to fill large amounts of memory with shellcode, hoping that the random location where control flow will land will be occupied by attacker code.

Attacks are not limited to hijacking function pointers fields in heap objects. Unfortunately, object oriented programming with manual memory management is inviting use-after-free attacks: C++ objects contain pointers to virtual tables (`vtables`) used for resolving virtual functions. In turn, these `vtables` contain pointers to virtual functions of the object's class. Attackers can hijack the `vtable` pointers diverting virtual function calls made through dangling pointers to a bogus `vtable`, and execute attacker code. Such `vtable` pointers abound in the heap memory of C++ programs.

Attackers may have to overcome an obstacle: the `vtable` pointer in a freed object is often aligned with the `vtable` pointer in the new object occupying the freed object's memory. This situation is likely, because the `vtable` pointer typically occupies the first word of an object's memory, and hence will be likely aligned with the `vtable` pointer of a new object allocated in its place right after the original object was freed. The attack is disrupted because the attacker lacks sufficient control over the new object's `vtable` pointer value that is maintained by the language runtime, and always points to a genuine, even if belonging to the wrong type, `vtable`, rather than arbitrary, attacker-controlled data. Attackers may overcome this problem by exploiting objects using multiple inheritance that have multiple `vtable` pointers located at various offsets, or objects derived from a base class with no virtual functions that do not have `vtable` pointers at offset zero, or by manipulating the heap to achieve an exploitable alignment through an appropriate sequence of allocations and deallocations. We will see that our defense prevents attackers from achieving such exploitable alignments.

Attacks are not limited to subverting control flow; they can also hijack data fields [7]. Hijacked data pointers, for instance, can be exploited to overwrite other targets, including function pointers, indirectly: if a program writes through a data pointer field of a deallocated object, an attacker controlling the memory contents of the deallocated object can divert the write to an arbitrary memory location. Other potential attacks include information leaks through reading the contents of a dangling pointer now pointing to sensitive information, and privilege escalation by hijacking data fields holding credentials.

Under certain memory allocator designs, dangling pointer bugs can be exploited without memory having to be reused by another object. Memory allocator metadata stored in free memory, such as pointers chaining free memory chunks into free lists, can play the role of the other object. When the deallocated object is referenced through a dangling pointer, the heap metadata occupying its memory will be interpreted as its fields. For example, a free list pointer may point to a chunk of free memory that contains leftover attacker data, such as a bogus `vtable`. Calling a virtual function through the dangling pointer would divert control to an arbitrary location of the attacker's choice. We must consider such attacks when designing a memory allocator to mitigate use-after-free vulnerabilities.

Finally, in all the above scenarios, attackers exploit reads through dangling pointers, but writes through a dangling pointer could also be exploited, by corrupting the object, or allocator metadata, now occupying the freed object's memory.

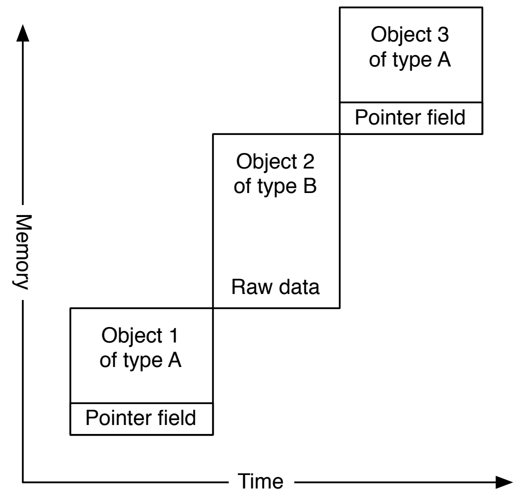

Figure 2: No memory reuse (very safe but expensive).

2.2 Naive Defense

A straight forward defense against use-after-free vulnerabilities that takes advantage of the abundant address space of modern 64-bit machines is avoiding any address space reuse. Excessive memory consumption can be avoided by reusing freed memory via the operating system's virtual memory mechanisms (*e.g.* relinquishing physical memory using `madvise` with the `MADV_DONTNEED` option on Linux, or other OS specific mechanisms). This simple solution, illustrated in Figure 2, protects against all the attacks discussed in Section 2.1, but has three shortcomings.

First, address space will eventually be exhausted. By then, however, the memory allocator could wrap around and reuse the address space without significant risk.

The second problem is more important. Memory fragmentation limits the amount of physical memory that can be reused through virtual memory mechanisms. Operating systems manage physical memory in units of several Kilobytes in the best case, thus, each small allocation can hold back several Kilobytes of physical memory in adjacent free objects from being reused. In Section 4, we show that the memory overhead of this solution is too high.

Finally, this solution suffers from a high rate of system calls to relinquish physical memory, and attempting to reduce this rate by increasing the block size of memory relinquished with a single system call leads to even higher memory consumption.

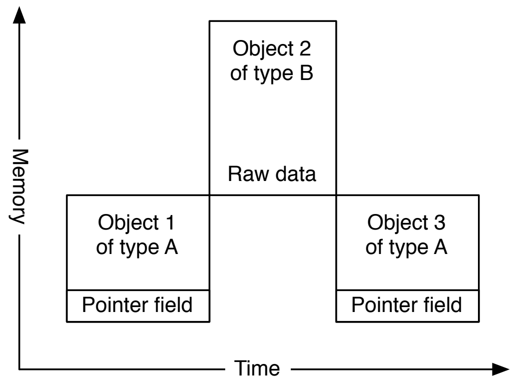

Figure 3: Type-safe memory reuse.

2.3 Type-Safe Memory Reuse

Type-safe memory reuse, proposed by Dhurjati *et al.* [9], allows some memory reuse while preserving type safety. It allows dangling pointers, but constrains them to point to objects of the same type and alignment. This way, dereferencing a dangling pointer cannot cause a type violation, rendering use-after-free bugs hard to exploit in practice. As illustrated in Figure 3, with type-safe memory reuse, memory formerly occupied by pointer fields cannot be reused for raw data, preventing attacks as the one in Figure 1.

Moreover, memory formerly occupied by pointer fields can only overlap with the corresponding pointer fields in objects of the same type. This means, for example, that a hijacked function pointer can only be diverted to some other function address used for the same field in a different object, precluding diverting function pointers to attacker injected code, and almost certainly thwarting return-to-libc [20] attacks diverting function pointers to legitimate but suitable executable code in the process. More importantly, objects of the same type share `vtables` and their `vtable` pointers are at the same offsets, thus type-safe memory reuse completely prevents hijacking of `vtable` pointers. This is similar to the attacker constraint discussed in Section 2.1, where the old `vtable` pointer happens to be aligned with another `vtable` pointer, except that attackers are even more constrained now: they cannot exploit differences in inheritance relationships or evade the obstacle by manipulating the heap.

These cases cover generic exploitation techniques and attacks observed in the wild. The remaining attacks are less practical but may be exploitable in some cases, depending on the application and its use of data. Some constraints may still be useful; for example, attacks that hijack data pointers are constrained to only access memory in the corresponding field of another object of the same type. In some cases, this may prevent dangerous corruption or data leakage. However, reusing memory of an object's data fields for another instance of the same type may still enable attacks, including privilege escalation attacks, *e.g.* when data structures holding credentials or access control information for different users are overlapped in time. Another potential exploitation avenue are inconsistencies in the program's data structures that may lead to other memory errors, *e.g.* a buffer may become inconsistent with its size stored in a different object when either is accessed through a dangling pointer. Interestingly, this inconsistency can be detected if spatial protection mechanisms, such as bounds checking, are used in tandem.

3 Cling Memory Allocator

The Cling memory allocator is a drop-in replacement for `malloc` designed to satisfy three requirements: *(i)* it does not reuse free memory for its metadata, *(ii)* only allows address space reuse among objects of the same type and alignment, and *(iii)* achieves these without sacrificing performance. Cling combines several solutions from existing memory allocators to achieve its requirements.

3.1 Out-of-Band Heap Metadata

The first requirement protects against use-after-free vulnerabilities with dangling pointers to free, not yet reallocated, memory. As we saw in Section 2.1, if the memory allocator uses freed memory for metadata, such as free list pointers, these allocator metadata can be interpreted as object fields, *e.g.* `vtable` pointers, when free memory is referenced through a dangling pointer.

Memory allocator designers have considered using out-of-band metadata before, because attackers targeted in-band heap metadata in several ways: attacker controlled data in freed objects can be interpreted as heap-metadata through double-free vulnerabilities, and heap-based overflows can corrupt allocator metadata adjacent to heap-based buffers. If the allocator uses corrupt heap metadata during its linked list operations, attackers can write an arbitrary value to an arbitrary location.

Although out-of-band heap metadata can solve these problems, some memory allocators mitigate heap metadata corruption without resorting to this solution. For example, attacks corrupting heap metadata can be addressed by detecting the use of corrupted metadata with sanity checks on free list pointers before unlinking a free chunk or using heap canaries [19] to detect corruption due to heap-based buffer overflows. In some cases, corruption can be prevented in the first place, *e.g.* by detecting attempts to free objects already in a free list. These techniques avoid the memory overhead of out-of-band metadata, but are insufficient for preventing use-after-free vulnerabilities, where no corruption of heap metadata takes place.

An approach to address this problem in allocator designs reusing free memory for heap metadata is to ensure that these metadata point to invalid memory if interpreted as pointers by the application. Merely randomizing the metadata by XORing with a secret value may not be sufficient in the face of heap spraying. One option is setting the top bit of every metadata word to ensure it points to protected kernel memory, raising a hardware fault if the program dereferences a dangling pointer to heap metadata, while the allocator would flip the top bit before using the metadata. However, it is still possible that the attacker can tamper with the dangling pointer before dereferencing it. This approach may be preferred when modifying an existing allocator design, but for Cling, we chose to keep metadata out-of-band instead.

An allocator can keep its metadata outside deallocated memory using non-intrusive linked lists (`next` and `prev` pointers stored outside objects) or bitmaps. Non-intrusive linked lists can have significant memory overhead for small allocations, thus Cling uses a two-level allocation scheme where non-intrusive linked lists chain large memory chunks into free lists and small allocations are carved out of buckets holding objects of the same size class using bitmaps. Bitmap allocation schemes have been used successfully in popular memory allocators aiming for performance [10], so they should not pose an inherent performance limitation.

3.2 Type-Safe Address Space Reuse

The second requirement protects against use-after-free vulnerabilities where the memory pointed by the dangling pointer has been reused by some other object. As we saw in Section 2.3, constraining dangling pointers to objects within pools of the same type and alignment thwarts a large class of attacks exploiting use-after-free vulnerabilities, including all those used in real attacks. A runtime memory allocator, however, must address two challenges to achieve this. First, it must bridge the semantic gap between type information available to the compiler at compile time and memory allocation requests received at runtime that only specify the number of bytes to allocate. Second, it must address the memory overheads caused by constraining memory reuse within pools. Dhurjati *et al.* [9], who proposed type-safe memory reuse for security, preclude an efficient implementation without using a compile time pointer and region analysis.

To solve the first challenge, we observe that security is maintained even if memory reuse is over-constrained, *i.e.* several allocation pools may exist for the same type, as long as memory reuse across objects of *different* types is prevented. Another key observation is that in C/C++ programs, an allocation site typically allocates objects of a single type or arrays of objects of a single type, which can safely share a pool. Moreover, the allocation site is available to the allocation routines by inspecting their call stack. While different allocation sites may allocate objects of the same type that could also safely share the same pool, Cling's inability to infer this could only affect performance—not security. Section 4 shows that in spite of this pessimization, acceptable performance is achieved.

The immediate caller of a memory allocation routine can be efficiently retrieved from the call stack by inspecting the saved return address. However, multiple tail-call optimizations in a single routine, elaborate control flow, and simple wrappers around allocation routines may obscure the true allocation site. The first two issues are sufficiently rare to not undermine the security of the scheme in general. These problems are elaborated in Section 3.6, and ways to address simple wrappers are described in Section 3.5.

A further complication, illustrated in Figure 4, is caused by array allocations and the lack of knowledge of array element sizes. As discussed, all new objects must be aligned to previously allocated objects, to ensure their fields are aligned one to one. This requirement also applies to array elements. Figure 4, however, illustrates that this constraint can be violated if part of the memory previously used by an array is subsequently reused by an allocation placed at an arbitrary offset relative to

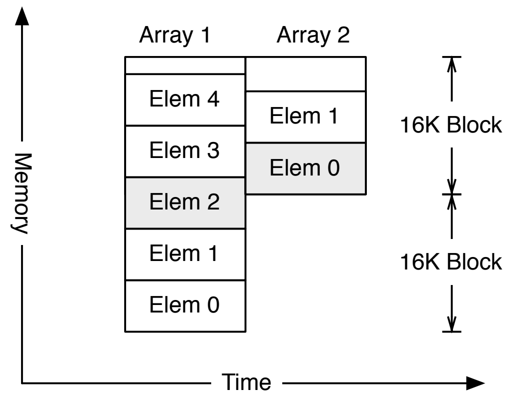

Figure 4: Example of unsafe reuse of array memory, even with allocation pooling, due to not preserving allocation offsets.

the start of the old allocation. Reusing memory from a pool dedicated to objects of the same type is not sufficient for preventing this problem. Memory reuse must also preserve offsets within allocated memory. One solution is to always reuse memory chunks at the same offset within all subsequent allocations. A more constraining but simpler solution, used by Cling, is to allow memory reuse only among allocations of the same size-class, thus ensuring that previously allocated array elements will be properly aligned with array elements subsequently occupying their memory.

This constraint also addresses the variable sized struct idiom, where the final field of a structure, such the following one, is used to access additional, variable size memory allocated at the end of the structure:

```
struct {
    void (*fp)();
    int len;
    char buffer[1];
};
```

By only reusing memory among instances of such structures that fall into the same size-class, and always aligning such structures at the start of this memory, Cling prevents the structure's fields, *e.g.* the function pointer `fp` in this example, from overlapping after their deallocation with buffer contents of some other object of the same type.

The second challenge is to address the memory overhead incurred by pooling allocations. Dhurjati *et al.* [8] observe that the worst-case memory use increase for a program with N pools would be roughly a factor of $N-1$: when a program first allocates data of type A, frees all of it, then allocates data of type B, frees all of it, and so on. This situation is even worse for Cling, because it has one pool per size-class per allocation site, instead of just one pool per type.

The key observation to avoid excessive memory overhead is that physical memory, unlike address space, can be safely reused across pools. Cling borrows ideas from previous memory allocators [11] designed to manage physical memory in blocks (via `mmap`) rather than monotonically growing the heap (via `sbrk`). These allocators return individual blocks of memory to the operating system as soon as they are completely free. This technique allows Cling to reuse blocks of memory across different pools.

Cling manages memory in blocks of 16K bytes, satisfying large allocations using contiguous ranges of blocks directly, while carving smaller allocations out of homogeneous blocks called buckets. Cling uses an OS primitive (*e.g.* `madvise`) to inform the OS it can reuse the physical memory of freed blocks.

Deallocated memory accessed through a dangling pointer will either continue to hold the data of the intended object, or will be zero-filled by the OS, triggering a fault if a pointer field stored in it is dereferenced. It is also possible to page protect address ranges after relinquishing their memory (*e.g.* using mechanisms like `mprotect` on top of `madvise`).

Cling does not suffer from fragmentation as the naive scheme described in Section 2.2, because it allows immediate reuse of small allocations' memory within a pool. Address space consumption is also more reasonable: it is proportional to the number of allocation sites in the program, so it does not leak over time as in the naive solution, and is easily manageable in modern 64-bit machines.

3.3 Heap Organization

Cling's main heap is divided into blocks of 16K bytes. As illustrated in Figure 5, a smaller address range, dubbed the meta-heap, is reserved for holding block descriptors, one for each 16K address space block. Block descriptors contain fields for maintaining free lists of block ranges, storing the size of the block range, associating the block with a pool, and pointers to metadata for blocks holding small allocations. Metadata for block ranges are only set for the first block in the range—the head block. When address space is exhausted and the heap is grown, the meta-heap is grown correspondingly. The purpose of this meta-heap is to keep heap metadata separate, allowing reuse of the heap's physical memory previously holding allocated data without discarding its metadata stored in the meta-heap.

While memory in the heap area can be relinquished using `madvise`, metadata about address space must be kept in the meta-heap area, thus contributing to the memory overhead of the scheme. This overhead is small. A block descriptor can be under 32 bytes in the current implementation, and with a block size of 16K, this corresponds to memory overhead less than 0.2% of the address space used, which is small enough for the address space usage observed in our evaluation. Moreover, a hashtable could be employed to further reduce this overhead if necessary.

Both blocks and block descriptors are arranged in corresponding linear arrays, as illustrated in Figure 5, so Cling can map between address space blocks and their corresponding block descriptors using operations on their addresses. This allows Cling to efficiently recover the appropriate block descriptor when deallocating memory.

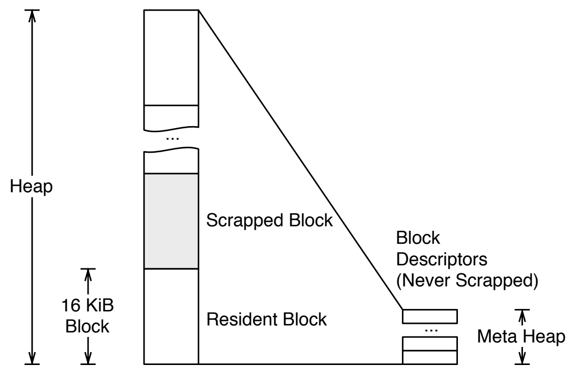

Figure 5: Heap comprised of blocks and meta-heap of block descriptors. The physical memory of deallocated blocks can be scrapped and reused to back blocks in other pools.

Cling pools allocations based on their allocation site. To achieve this, Cling's public memory allocation routines (*e.g.* `malloc` and `new`) retrieve their call site using the return address saved on the stack. Since Cling's routines have complete control over their prologues, the return address can always be retrieved reliably and efficiently (*e.g.* using the `__builtin_return_address` GCC primitive). At first, this return address is used to distinguish between memory allocation sites. Section 3.5 describes how to discover and unwind simple allocation routine wrappers in the program, which is necessary for obtaining a meaningful allocation site in those cases.

Cling uses a hashtable to map allocation sites to pools at runtime. An alternative design to avoid hash table lookups could be to generate a trampoline for each call site and rewrite the call site at hand to use its dedicated trampoline instead of directly calling the memory allocation routine. The trampoline could then call a version of the memory allocation routine accepting an explicit pool parameter. The hash table, however, was preferred because it is less intrusive and handles gracefully corner cases including calling `malloc` through a function pointer. Moreover, since this hash table is accessed frequently but updated infrequently, optimizations such as constructing perfect hashes can be applied in the future, if necessary.

Pools are organized around pool descriptors. The relevant data structures are illustrated in Figure 6. Each pool descriptor contains a table with free lists for block ranges. Each free list links together the head blocks of block ranges belonging to the same size-class (a power of two). These are blocks of memory that have been deallocated and are now reusable only within the pool. Pool descriptors also contain lists of blocks holding small allocations, called buckets. Section 3.4 discusses small object allocation in detail.

Initially, memory is not assigned to any pool. Larger allocations are directly satisfied using a power-of-two range of 16K blocks. A suitable free range is reused from the pool if possible, otherwise, a block range is allocated by incrementing a pointer towards the end of the heap, and it is assigned to the pool. If necessary, the heap is grown using a system call. When these large allocations are deallocated, they are inserted to the appropriate pool descriptor's table of free lists according to their size. The free list pointers are embedded in block descriptors, allowing the underlying physical memory for the block to be relinquished using `madvise`.

3.4 Small Allocations

Allocations less than 8K in size (half the block size) are stored in slots inside blocks called buckets. Pool descriptors point to a table with entries to manage buckets for allocations belonging to the same size class. Size classes start from a minimum of 16 bytes, increase by 16 bytes up to 128 bytes, and then increase exponentially up to the maximum of 8K, with 4 additional classes in between each pair of powers-of-two. Each bucket is associated with a free slot bitmap, its element size, and a bump pointer used for fast allocation when the block is first used, as described next.

Using bitmaps for small allocations seems to be a design requirement for keeping memory overhead low without reusing free memory for allocator metadata, so it is critical to ensure that bitmaps are efficient compared to free-list based implementations. Some effort has been put into making sure Cling uses bitmaps efficiently. Cling borrows ideas from reaps [5] to avoid bitmap scanning when many objects are allocated from an allocation site in bursts. This case degenerates to just bumping a pointer to allocate consecutive memory slots. All empty buckets are initially used in bump mode, and

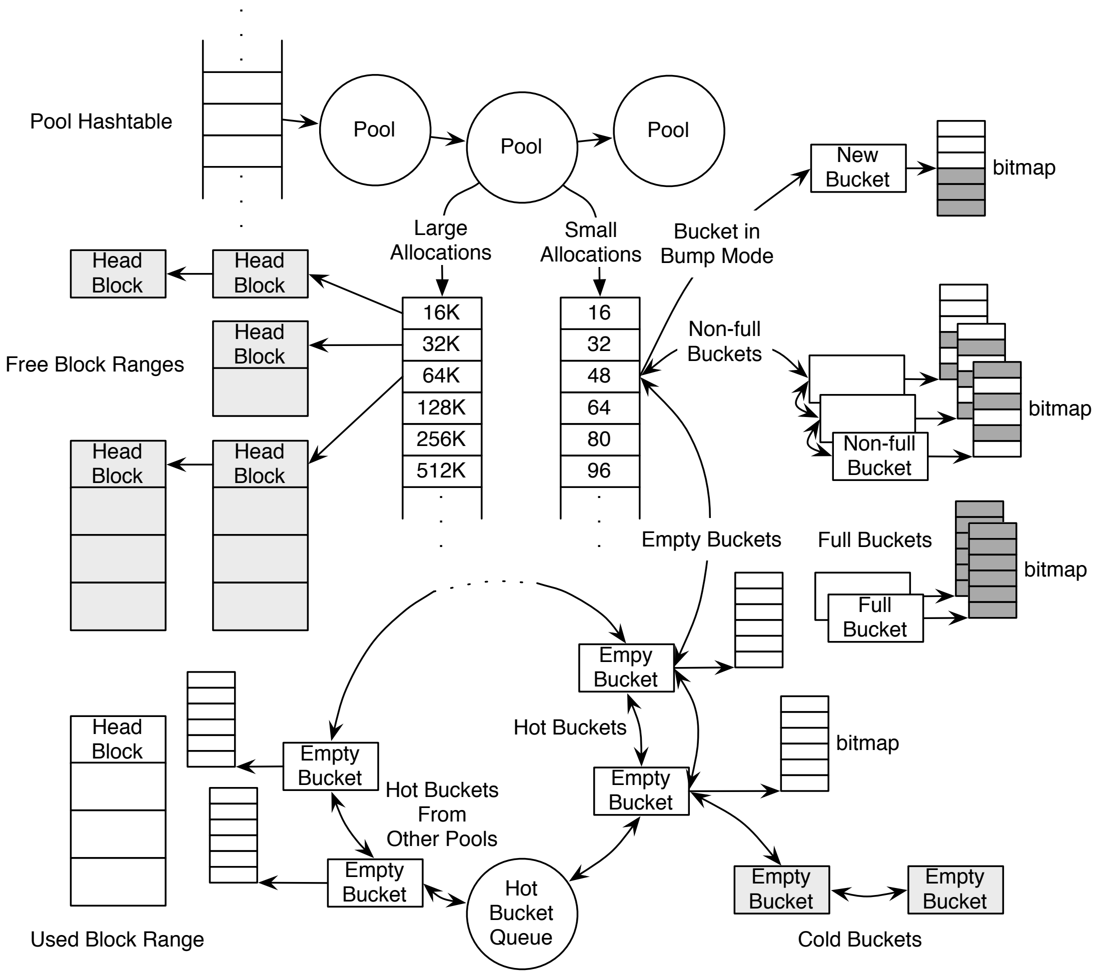

Figure 6: Pool organization illustrating free lists of blocks available for reuse within the pool and the global hot bucket queue that delays reclamation of empty bucket memory. Linked list pointers are not stored inside blocks, as implied by the figure, but rather in their block descriptors stored in the meta-heap. Blocks shaded light gray have had their physical memory reclaimed.

stay in that mode until the bump pointer reaches the end of the bucket. Memory released while in bump mode is marked in the bucket's bitmap but is not used for satisfying allocation requests while the bump pointer can be used.

A pool has at most one bucket in bump mode per size class, pointed by a field of the corresponding table entry, as illustrated in Figure 6. Cling first attempts to satisfy an allocation request using that bucket, if available. Buckets maintain the number of freed elements in a counter. A bucket whose bump pointer reaches the end of the bucket is unlinked from the table entry and, if the counter indicates it has free slots, inserted into a list of non-full buckets. If no bucket in bump mode is available, Cling attempts to use the first bucket from this list, scanning its bitmap to find a free slot. If the counter indicates the bucket is full after an allocation request, the bucket is unlinked from the list of non-full buckets, to avoid obstracting allocations.

Conversely, if the counter of free elements is zero prior to a deallocation, the bucket is re-inserted into the list of non-full buckets. If the counter indicates that the bucket is completely empty after deallocation, it is inserted to a list of empty buckets queuing for memory reuse. This applies even for buckets in bump mode (and was important for keeping memory overhead low). This list of empty buckets is consulted on allocation if there is neither a bucket in bump mode, nor a non-full bucket. If this list is also empty, a new bucket is created using fresh address space, and initialized in bump mode.

Empty buckets are inserted into a global queue of hot buckets, shown at the bottom of Figure 6. This queue has a configurable maximum size (10% of non-empty buckets worked well in our experiments). When the queue size threshold is reached after inserting an empty bucket to the head of the queue, a hot bucket is removed from the tail of the queue, and becomes cold: its bitmap is deallocated, and its associated 16K of memory reused via an `madvise` system call. If a cold bucket is encountered when allocating from the empty bucket list of a pool, a new bitmap is allocated and initialized. The hot bucket queue is important for reducing the number of system calls by trading some memory overhead, controllable through the queue size threshold.

3.5 Unwinding Malloc Wrappers

Wrappers around memory allocation routines may conceal real allocation sites. Many programs wrap `malloc` simply to check its return value or collect statistics. Such programs could be ported to Cling by making sure that the few such wrappers call macro versions of Cling's allocation routines that capture the real allocation site, *i.e.* the wrapper's call site. That is not necessary, however, because Cling can detect and handle many such wrappers automatically, and recover the real allocation site by unwinding the stack. This must be implemented carefully because stack unwinding is normally intended for use in slow, error handing code paths.

To detect simple allocation wrappers, Cling initiates a probing mechanism after observing a single allocation site requesting multiple allocation sizes. This probing first uses a costly but reliable unwind of the caller's stack frame (using `libunwind`) to discover the stack location of the suspected wrapper function's return address. Then, after saving the original value, Cling overwrites the wrapper's return address on the stack with the address of a special assembler routine that will be interposed when the suspected wrapper returns. After Cling returns to the caller, and, in turn, the caller returns, the overwritten return address transfers control to the interposed routine. This routine compares the suspected allocation wrapper's return value with the address of the memory allocated by Cling, also saved when the probe was initiated. If the caller appears to return the address just returned by Cling, it is assumed to be a simple wrapper around an allocation function.

To simplify the implementation, probing is aborted if the potential wrapper function issues additional allocation requests before returning. This is not a problem in practice, because simple `malloc` wrappers usually perform a single allocation. Moreover, a more thorough implementation can easily address this.

The probing mechanism is only initiated when multiple allocation sizes are requested from a single allocation site, potentially delaying wrapper identification. It is unlikely, however, that an attacker could exploit this window of opportunity in large programs. Furthermore, this rule helps prevent misidentifying typical functions encapsulating the allocation and initialization of objects of a single type, because these request objects of a single size. Sometimes, such functions allocate arrays of various sizes, and can be misidentified. Nevertheless, these false positives are harmless for security; they only introduce more pools that affect performance by overconstraining allocation, and the performance impact in our benchmarks was small.

Similarly, the current implementation identifies functions such as `strdup` as allocation wrappers. While we could safely pool their allocations (they are of the same type), the performance impact in our benchmarks was again small, so we do not handle them in any special way.

While this probing mechanism handles well the common case of `malloc` wrappers that return the allocated memory through their function return value, it would not detect a wrapper that uses some other mechanism to return the memory, such as modifying a pointer argument passed to the wrapper by reference. Fortunately, such `malloc` wrappers are unusual.

Allocation sites identified as potential wrappers through this probing mechanism are marked as such in the hashtable mapping allocation site addresses to their pools, so Cling can unwind one more stack level to get the real allocation site whenever allocation requests from such an allocation site are encountered, and associate it with a distinct pool.

Stack unwinding is platform specific and, in general, expensive. In 32-bit x86 systems, the frame pointer register `ebp` links stacks frames together, making unwinding reasonably fast, but this register may be re-purposed in optimized builds. Heuristics can still be used with optimized code, *e.g.* looking for a value in the stack that points into the text segment, but they are slower. Data-driven stack unwinding on 64-bit AMD64 systems is more reliable but, again, expensive. Cling uses the `libunwind` library to encapsulate platform specific details of stack unwinding, but caches the stack offset of wrappers' return addresses to allow fast unwinding when possible, as described next, and gives up unwinding if not.

Care must be taken when using a cached stack offset to retrieve the real allocation site, because the cached value may become invalid for functions with a variable frame size, *e.g.* those using `alloca`, resulting in the retrieval of a bogus address. To guard against this, whenever a *new* allocation site is encountered that was retrieved using a cached stack offset, a slow but reliable unwind (using `libunwind`) is performed to confirm the allocation site's validity. If the check fails, the wrapper must have a variable frame size, and Cling falls back to allocating all memory requested through that wrapper from a single pool. In practice, typical malloc wrappers are simple functions with constant frame sizes.

3.6 Limitations

Cling prevents vtable hijacking, the standard exploitation technique for use-after-free vulnerabilities, and its constraints on function and data pointers are likely to prevent their exploitation, but it may not be able to prevent use-after-free attacks targeting data such as credentials and access control lists stored in objects of a single type. For example, a dangling pointer that used to point to the credentials of one user may end up pointing to the credentials of another user.

Another theoretical attack may involve data structure inconsistencies, when accessed through dangling pointers. For example, if a buffer and a variable holding its length are in separate objects, and one of them is read through a dangling pointer accessing an unrelated object, the length variable may be inconsistent with the actual buffer length, allowing dangerous bound violations. Interestingly, this can be detected if Cling is used in conjunction with a defense offering spatial protection.

Cling relies on mapping allocation sites to object types. A program with contrived flow of control, however, such as in the following example, would obscure the type of allocation requests:

```
int size = condition ? sizeof( ↩
  struct A) : sizeof(struct B);
void *obj = malloc(size);
```

Fortunately, this situation is less likely when allocating memory using the C++ operator new that requires a type argument.

A similar problem occurs when the allocated object is a union: objects allocated at the same program location may still have different types of data at the same offset.

Tail-call optimizations can also obscure allocation sites. Tail-call optimization is applicable when the call to malloc is the last instruction before a function returns. The compiler can then replace the call instruction with a simple control-flow transfer to the allocation routine, avoiding pushing a return address to the stack. In this case, Cling would retrieve the return address of the function calling malloc. Fortunately, in most cases where this situation might appear, using the available return address still identifies the allocation site uniquely.

Cling cannot prevent unsafe reuse of stack allocated objects, for example when a function erroneously returns a pointer to a local variable. This could be addressed by using Cling as part of a compiler-based solution, by moving dangerous (*e.g.* address taken) stack based variables to the heap at compile time.

Custom memory allocators are a big concern. They allocate memory in huge chunks from the system allocator, and chop them up to satisfy allocation requests for individual objects, concealing the real allocation sites of the program. Fortunately, many custom allocators are used for performance when allocating many objects of a *single* type. Thus, pooling such custom allocator's requests to the system allocator, as done for any other allocation site, is sufficient to maintain type-safe memory reuse. It is also worth pointing that roll-your-own general purpose memory allocators have become a serious security liability due to a number of exploitable memory management bugs beyond use-after-free (invalid frees, double frees, and heap metadata corruption in general). Therefore, using a custom allocator in new projects is not a decision to be taken lightly.

Usability in 32-bit platforms with scarce address space is limited. This is less of a concern for high-end and future machines. If necessary, however, Cling can be combined with a simple conservative collector that scans all words in used physical memory blocks for pointers to used address space blocks. This solution avoids some performance and compatibility problems of conservative garbage collection by relying on information about explicit deallocations. Once address space is exhausted, only memory that is in use needs to be scanned and any 16K block of freed memory that is not pointed by any word in the scanned memory can be reused. The chief compatibility problem of conservative garbage collection, namely hidden pointers (manufactured pointers invisible to the collector), cannot cause premature deallocations, because only explicitly deallocated memory would be garbage collected in this scheme. Nevertheless, relying on the abundant address space of modern machines instead, is more attractive, because garbage collection may introduce unpredictability or expose the program to attacks using hidden dangling pointers.

3.7 Implementation

Cling comes as a shared library providing implementations for the malloc and the C++ operator new allocation interfaces. It can be preloaded with platform specific mechanisms (*e.g.* the LD_PRELOAD environment variable on most Unix-based systems) to override the system's memory allocation routines at program load time.

4 Experimental Evaluation

4.1 Methodology

We measured Cling's CPU, physical memory, and virtual address space overheads relative to the default GNU libc memory allocator on a 2.66GHz Intel Core 2 Q9400

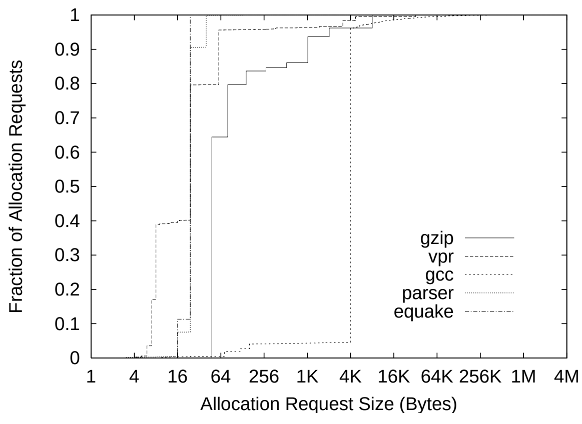

Figure 7: Cumulative distribution function of memory allocation sizes for `gzip`, `vpr`, `gcc`, `parser`, and `equake`.

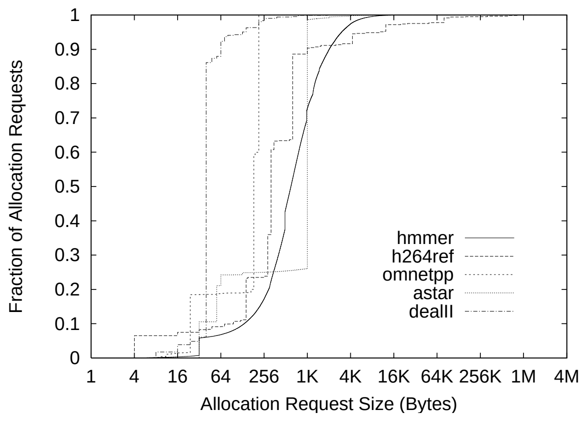

Figure 9: Cumulative distribution function of memory allocation sizes for `hmmer`, `h264ref`, `omnetpp`, `astar`, and `dealII`.

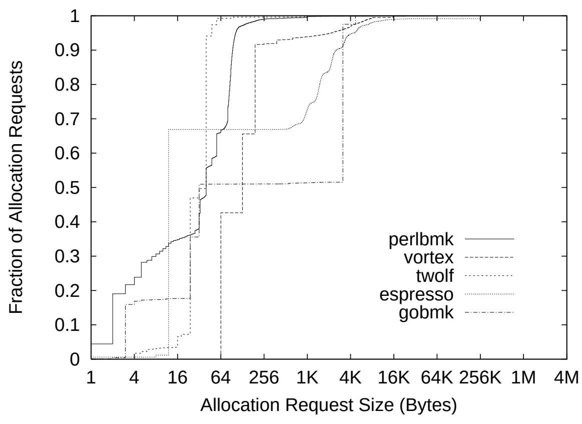

Figure 8: Cumulative distribution function of memory allocation sizes for `perlbmk`, `vortex`, `twolf`, `espresso`, and `gobmk`.

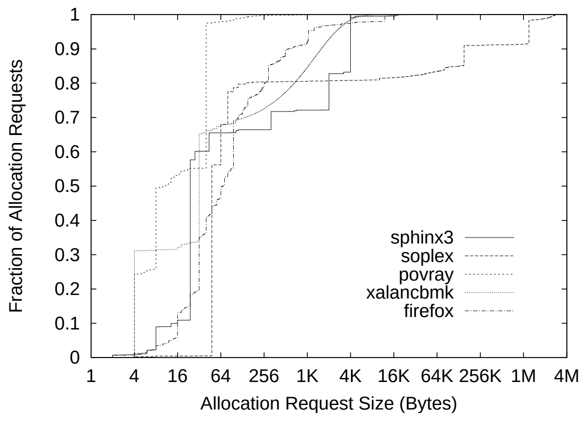

Figure 10: Cumulative distribution function of memory allocation sizes for `sphinx3`, `soplex`, `povray`, `xalancbmk`, and Firefox.

CPU with 4GB of RAM, running `x86_64` GNU/Linux with a version 2.6 Linux kernel. We also measured two variations of Cling: without wrapper unwinding and using a single pool.

We used benchmarks from the SPEC CPU 2000 and (when not already included in CPU 2000) 2006 benchmark suites [22]. Programs with few allocations and deallocations have practically no overhead with Cling, thus we present results for SPEC benchmarks with at least 100,000 allocation requests. We also used `espresso`, an allocation intensive program that is widely used in memory management studies, and is useful when comparing against related work. Finally, in addition to CPU bound benchmarks, we also evaluated Cling with a current version of the Mozilla Firefox web browser. Web browsers like Firefox are typical attack targets for use-after-free exploits via malicious web sites; moreover, unlike many benchmarks, Firefox is an application of realistic size and running time.

Some programs use custom allocators, defeating Cling's protection and masking its overhead. For these experiments, we disabled a custom allocator implementation in `parser`. The `gcc` benchmark also uses a custom allocation scheme (`obstack`) with different semantics from `malloc` that cannot be readily disabled. We include it to contrast its allocation size distribution with those of other benchmarks. Recent versions of Firefox also use a custom allocator [10] that was disabled by compiling from source with the `--disable-jemalloc` configuration option.

The SPEC programs come with prescribed input data. For `espresso`, we generated a uniformly random input file with 15 inputs and 15 outputs, totalling 32K lines. For Firefox, we used a list of 200 websites retrieved from our browsing history, and replayed it using the `-remote`

	Allocation Sites			Allocation Requests		Deallocation Requests	
Benchmark	Not Wrappers	Wrappers	Unwound	Small	Large	Small	Large
CPU2000							
gzip	3	0	0	419,724	16,483	419,724	16,463
vpr	11	2	59	107,184	547	103,390	42
gcc	5	1	66	194,871	4,922	166,317	4,922
parser	218	3	3	787,695,542	46,532	787,523,051	46,532
equake	31	0	0	1,335,048	19	0	0
perlbmk	10	3	90	31,399,586	33,088	30,704,104	32,732
vortex	5	0	0	4,594,278	28,094	4,374,712	26,373
twolf	3	1	129	574,552	15	492,722	5
CPU2006							
gobmk	50	5	15	621,144	20	621,109	0
hmmer	8	4	107	2,405,928	10,595	2,405,928	10,595
dealII	285	0	0	151,324,612	7,701	151,324,610	7,701
sphinx3	25	2	6	14,160,472	64,086	13,959,978	63,910
h264ref	342	0	0	168,634	9,145	168,631	9,142
omnetpp	158	1	17	267,167,577	895	267,101,325	895
soplex	285	6	25	190,986	44,959	190,984	44,959
povray	44	0	0	2,414,082	268	2,413,942	268
astar	102	0	0	4,797,794	2,161	4,797,794	2,161
xalancbmk	304	1	1	135,037,352	118,205	135,037,352	118,205
Other							
espresso	49	7	14	3,877,784	77,711	3,877,783	77,711
firefox	2101	51	595	22,579,058	464,565	22,255,963	464536

Table 1: Memory allocation sites and requests in benchmarks and Firefox browser.

option to direct a continuously running Firefox instance under measurement to a new web site every 10 seconds.

We report memory consumption using information obtained through the `/proc/self/status` Linux interface. When reporting physical memory consumption, the sum of the `VmRSS` and `VmPTE` fields is used. The latter measures the size of the page tables used by the process, which increases with Cling due to the larger address space. In most cases, however, it was still very small in absolute value. The `VmSize` field is used to measure address space size. The `VmPeak` and `VmHWM` fields are used to obtain peak values for the `VmSize` and `VmRSS` fields respectively.

The reported CPU times are averages over three runs with small variance. CPU times are not reported for Firefox, because the experiment was IO bound with significant variance.

4.2 Benchmark Characterization

Figures 7–10 illustrate the size distribution of allocation requests made by any given benchmark running with their respective input data. We observe that most benchmarks request a wide range of allocation sizes, but the `gcc` benchmark that uses a custom allocator mostly requests memory in chunks of 4K.

Table 1 provides information on the number of static allocation sites in the benchmarks and the absolute number of allocation and deallocation requests at runtime. For allocation sites, the first column is the number of allocation sites that are not wrappers, the second column is the number of allocation sites that are presumed to be in allocation routine wrappers (such as `safe_malloc` in `twolf`, `my_malloc` in `vpr`, and `xmalloc` in `gcc`), and the third column is the number of call sites of these wrappers, that have to be unwound. We observe that Firefox has an order of magnitude more allocation sites than the rest.

The number of allocation and deallocation requests for small (less than 8K) and large allocations are reported separately. The vast majority of allocation requests are for small objects and thus the performance of the bucket allocation scheme is crucial. In fact, no attempt was made to optimize large allocations in this work.

4.3 Results

Table 2 tabulates the results of our performance measurements. We observe that the runtime overhead is modest even for programs with a higher rate of allocation and deallocation requests. With the exception of `espresso` (16%), `parser` (12%), and `dealII` (8%), the overhead is less than 2%. Many other benchmarks with few allocation and deallocation requests, not presented here, have even less overhead—an interesting benefit of this approach, which, unlike solutions interposing on memory accesses, does not tax programs not making heavy use of dynamic memory.

In fact, many benchmarks with a significant number of allocations run faster with Cling. For example `xalancbmk`, a notorious allocator abuser, runs 25% faster. In many cases we observed that by tuning allo-

Benchmark	Execution time				Peak memory usage				Peak VM usage	
	Orig. (Sec.)	Cling Ratio			Orig. (MiB)	Cling Ratio			Orig. (MiB)	Cling Ratio
		Pools	No Unwind	No Pools		Pools	No Unwind	No Pools		Pools
CPU2000										
gzip	95.7	**1.00**	1.00	1.00	181.91	**1.00**	1.00	1.00	196.39	1.10
vpr	76.5	**1.00**	0.99	0.99	48.01	**1.06**	1.06	1.06	62.63	1.54
gcc	43.29	**1.01**	1.01	1.01	157.05	**0.98**	0.98	0.98	171.42	1.21
parser	152.6	**1.12**	1.08	1.05	21.43	**1.14**	1.13	1.05	35.99	2.26
equake	47.3	**0.98**	1.00	0.99	49.85	**0.99**	0.99	0.99	64.16	1.14
perlbmk	68.18	**1.02**	0.99	1.00	132.47	**0.96**	0.95	0.95	146.69	1.16
vortex	72.19	**0.99**	0.99	0.99	73.09	**0.91**	0.91	0.91	88.18	1.74
twolf	101.31	**1.01**	1.00	1.00	6.85	**0.93**	0.91	0.90	21.15	1.19
CPU2006										
gobmk	628.6	**1.00**	1.0	1.00	28.96	**1.01**	1.00	1.00	44.69	1.64
hmmer	542.15	**1.02**	1.02	1.01	25.75	**1.02**	1.01	1.01	40.31	1.79
dealII	476.74	**1.08**	1.07	1.06	793.39	**1.02**	1.02	1.02	809.46	1.70
sphinx3	1143.6	**1.00**	1.00	0.99	43.45	**1.01**	1.01	1.01	59.93	1.37
h264ref	934.71	**1.00**	1.01	1.01	64.54	**0.97**	0.97	0.96	80.18	1.52
omnetpp	573.7	**0.83**	0.83	0.87	169.58	**0.97**	0.97	0.97	183.45	1.03
soplex	524.01	**1.01**	1.01	1.01	421.8	**1.27**	1.27	1.27	639.51	2.31
povray	272.54	**1.00**	1.00	0.99	4.79	**1.33**	1.33	1.29	34.1	0.77
astar	656.09	**0.93**	0.93	0.92	325.77	**0.94**	0.94	0.94	345.51	1.56
xalancbmk	421.03	**0.75**	0.75	0.77	419.93	**1.03**	1.03	1.14	436.54	1.45
Other										
espresso	25.21	**1.16**	1.07	1.10	4.63	**1.13**	1.06	1.02	19.36	2.08

Table 2: Experimental evaluation results for the benchmarks.

cator parameters such as the block size and the length of the hot bucket queue, we were able to trade memory for speed and vice versa. In particular, with different block sizes, xalancbmk would run twice as fast, but with a memory overhead around 40%.

In order to factor out the effects of allocator design and tuning as much as possible, Table 2 also includes columns for CPU and memory overhead using Cling with a single pool (which implies no unwinding overhead as well). We observe that in some cases Cling with a single pool is faster and uses less memory than the system allocator, hiding the non-zero overheads of pooling allocations in the full version of Cling. On the other hand, for some benchmarks with higher overhead, such as dealII and parser, some of the overhead remains even without using pools. For these cases, both slow and fast, it makes sense to compare the overhead against Cling with a single pool. A few programs, however, like xalancbmk, use more memory or run slower with a single pool. As mentioned earlier, this benchmark is quite sensitive to allocator tweaks.

Table 2 also includes columns for CPU and memory overhead using Cling with many pools but without unwinding wrappers. We observe that for espresso and parser, some of the runtime overhead is due to this unwinding.

Peak memory consumption was also low for most benchmarks, except for parser (14%), soplex (27%), povray (33%), and espresso (13%). Interestingly, for soplex and povray, this overhead is not because of allocation pooling: these benchmarks incur similar memory overheads when running with a single pool. In the case of soplex, we were able to determine that the overhead is due to a few large realloc requests, whose current implementation in Cling is suboptimal. The allocation intensive benchmarks parser and espresso, on the other hand, do appear to incur memory overhead due to pooling allocations. Disabling unwinding also affects memory use by reducing the number of pools.

The last two columns of Table 2 report virtual address space usage. We observe that Cling's address space usage is well within the capabilities of modern 64-bit machines, with the worst increase less than 150%. Although 64-bit architectures can support much larger address spaces, excessive address space usage would cost in page table memory. Interestingly, in all cases, the address space increase did not prohibit running the programs on 32-bit machines. Admittedly, however, it would be pushing up against the limits.

In the final set of experiments, we ran Cling with Firefox. Since, due to the size of the program, this is the most interesting experiment, we provide a detailed plot of memory usage as a function of time (measured in allocated Megabytes of memory), and we also compare against the naive solution of Section 2.2.

The naive solution was implemented by preventing Cling from reusing memory and changing the memory block size to 4K, which is optimal in terms of memory reuse. (It does increase the system call rate how-

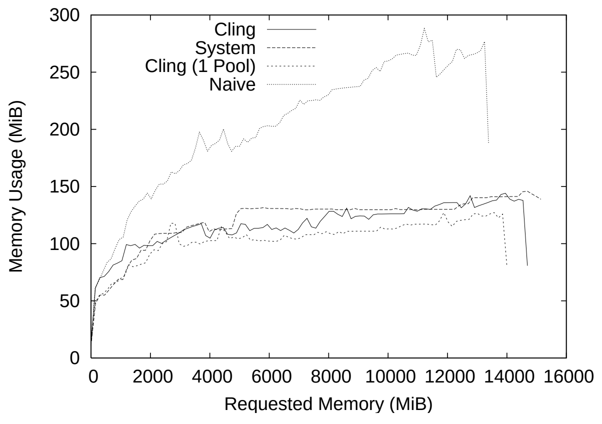

Figure 11: Firefox memory usage over time (measured in requested memory).

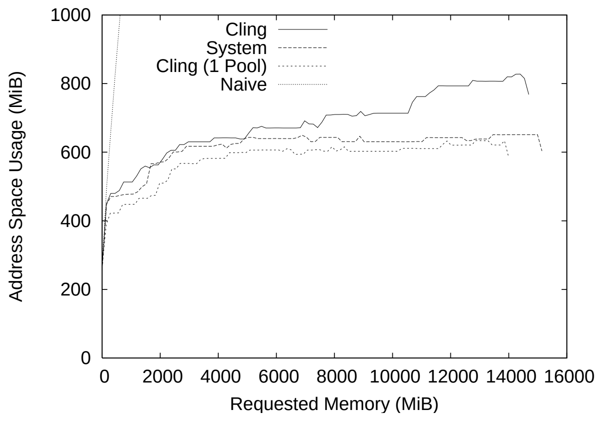

Figure 12: Firefox address space usage over time (measured in requested memory).

ever.) The naive solution could be further optimized by not using segregated storage classes, but this would not affect the memory usage significantly, as the overhead of rounding small allocation requests to size classes in Cling is at most 25%—and much less in practice.

Figure 11 graphs memory use for Firefox. We observe that Cling (with pools) uses similar memory to the system's default allocator. Using pools does incur some overhead, however, as we can see by comparing against Cling using a single pool (which is more memory efficient than the default allocator). Even after considering this, Cling's approach of safe address space reuse appears usable with large, real applications. We observe that Cling's memory usage fluctuates more than the default allocator's because it aggressively returns memory to the operating system. These graphs also show that the naive solution has excessive memory overhead.

Finally, Figure 12 graphs address space usage for Firefox. It illustrates the importance of returning memory to the operating system; without doing so, the scheme's memory overhead would be equal to its address space use. We observe that this implied memory usage with Firefox may not be prohibitively large, but many of the benchmarks evaluated earlier show that there are cases where it can be excessive. As for the address space usage of the naive solution, it quickly goes off the chart because it is linear with requested memory. The naive solution was also the only case where the page table overhead had a significant contribution during our evaluation: in this experiment, the system allocator used 0.99 MiB in page tables, Cling used 1.48 MiB, and the naive solution 19.43 MiB.

5 Related Work

Programs written in high-level languages using garbage collection are safe from use-after-free vulnerabilities, because the garbage collector never reuses memory while there is a pointer to it. Garbage collecting unsafe languages like C and C++ is more challenging. Nevertheless, conservative garbage collection [6] is possible, and can address use-after-free vulnerabilities. Conservative garbage collection, however, has unpredictable runtime and memory overheads that may hinder adoption, and is not entirely transparent to the programmer: some porting may be required to eliminate pointers hidden from the garbage collector.

DieHard [4] and Archipelago [16] are memory allocators designed to survive memory errors, including dangling pointer dereferences, with high probability. They can survive dangling pointer errors by preserving the contents of freed objects for a random period of time. Archipelago improves on DieHard by trading address space to decrease physical memory consumption. These solutions are similar to the naive solution of Section 2.2, but address some of its performance problems by eventually reusing memory. Security, however, is compromised: while their probabilistic guarantees are suitable for addressing reliability, they are insufficient against attackers who can adapt their attacks. Moreover, these solutions have considerable runtime overhead for allocation intensive applications. DieHard (without its replication feature) has 12% average overhead but up to 48.8% for `perlbmk` and 109% for `twolf`. Archipelago has 6% runtime overhead across a set of server applications with low allocation rates and few live objects, but the allocation intensive `espresso` benchmark runs 7.32 times slower than using the GNU libc allocator. Cling offers deterministic protection against dangling pointers (but not spatial violations), with significantly lower overhead (*e.g.* 16% runtime overhead for the allocation intensive `espresso` benchmark) thanks to allowing type-safe reuse within pools.

Dangling pointer accesses can be detected using compile-time instrumentation to interpose on every memory access [3, 24]. This approach guarantees complete temporal safety (sharing most of the cost with spatial safety), but has much higher overhead than Cling.

Region-based memory management (*e.g.* [14]) is a language-based solution for safe and efficient memory management. Object allocations are maintained in a lexical stack, and are freed when the enclosing block goes out of scope. To prevent dangling pointers, objects can only refer to other objects in the same region or regions higher up the stack. It may still have to be combined with garbage collection to address long-lived regions. Its performance is better than using garbage collection alone, but it is not transparent to programmers.

A program can be manually modified to use reference-counted smart pointers to prevent reusing memory of objects with remaining references. This, however, requires major changes to application code. HeapSafe [12], on the other hand, is a solution that applies reference counting to legacy code automatically. It has reasonable overhead over a number of CPU bound benchmarks (geometric mean of 11%), but requires recompilation and some source code tweaking.

Debugging tools, such as Electric Fence, use a new virtual page for each allocation of the program and rely on page protection mechanisms to detect dangling pointer accesses. The physical memory overheads due to padding allocations to page boundaries make this approach impractical for production use. Dhurjati *et al.* [8] devised a mechanism to transform memory overhead to address space overhead by wrapping the memory allocator and returning a pointer to a dedicated new virtual page for each allocation but mapping it to the physical page used by the original allocator. The solution's runtime overhead for Unix servers is less than 4%, and for other Unix utilities less than 15%, but incurs up to 11× slowdown for allocation intensive benchmarks.

Interestingly, type-safe memory reuse (dubbed type-stable memory management [13]) was first used to simplify the implementation of non-blocking synchronization algorithms by preventing type errors during speculative execution. In that case, however, it was not applied indiscriminately, and memory could be safely reused after some time bound; thus, performance issues addressed in this work were absent.

Dynamic pool allocation based on allocation site information retrieved by `malloc` through the call stack has been used for dynamic memory optimization [25]. That work aimed to improve performance by laying out objects allocated from the same allocation site consecutively in memory, in combination with data prefetching instructions inserted into binary code.

Dhurjati *et al.* [9] introduced type-homogeneity as a weaker form of temporal memory safety. Their solution uses automatic pool allocation at compile-time to segregate objects into pools of the same type, only reusing memory within pools. Their approach is transparent to the programmer and preserves address space, but relies on imprecise, whole-program analysis.

WIT [2] enforces an approximation of memory safety. It thwarts some dangling pointer attacks by constraining writes and calls through hijacked pointer fields in structures accessed through dangling pointers. It has an average runtime overhead of 10% for SPEC benchmarks, but relies on imprecise, whole-program analysis.

Many previous systems only address the spatial dimension of memory safety (*e.g.* bounds checking systems like [15]). These can be complemented with Cling to address both spatial and temporal memory safety.

Finally, address space layout randomization (ASLR) and data execution prevention (DEP) are widely used mechanisms designed to thwart exploitation of memory errors in general, including use-after-free vulnerabilities. These are practical defenses with low overhead, but they can be evaded. For example, a non-executable heap can be bypassed with, so called, *return-to-libc* attacks [20] diverting control-flow to legitimate executable code in the process image. ASLR can obscure the locations of such code, but relies on secret values, which a lucky or determined attacker might guess. Moreover, buffer overreads [23] can be exploited to read parts of the memory contents of a process running a vulnerable application, breaking the secrecy assumptions of ASLR.

6 Conclusions

Pragmatic defenses against low-level memory corruption attacks have gained considerable acceptance within the software industry. Techniques such as stack canaries, address space layout randomization, and safe exception handling —thanks to their low overhead and transparency for the programmer— have been readily employed by software vendors. In particular, attacks corrupting metadata pointers used by the memory management mechanisms, such as invalid frees, double frees, and heap metadata overwrites, have been addressed with resilient memory allocator designs, benefiting many programs transparently. Similar in spirit, Cling is a pragmatic memory allocator modification for defending against use-after-free vulnerabilities that is readily applicable to real programs and has low overhead.

We found that many of Cling's design requirements could be satisfied by combining mechanisms from successful previous allocator designs, and are not inherently detrimental for performance. The overhead of mapping allocation sites to allocation pools was found acceptable in practice, and could be further addressed in future implementations. Finally, closer integration with the language by using compile-time libraries is possible, especially for C++, and can eliminate the semantic gap between the language and the memory allocator by forwarding type information to the allocator, increasing security and flexibility in memory reuse. Nevertheless, the current instantiation has the advantage of being readily applicable to a problem with no practical solutions.

Acknowledgments

We would like to thank Amitabha Roy for his suggestion of intercepting returning functions to discover potential allocation routine wrappers, Asia Slowinska for fruitful early discussions, and the anonymous reviewers for useful, to-the-point comments.

References

[1] AFEK, J., AND SHARABANI, A. Dangling pointer: Smashing the pointer for fun and profit. In *Black Hat USA Briefings* (Aug. 2007).

[2] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing memory error exploits with WIT. In *Proceedings of the IEEE Symposium on Security and Privacy* (Los Alamitos, CA, USA, 2008), IEEE Computer Society, pp. 263–277.

[3] AUSTIN, T. M., BREACH, S. E., AND SOHI, G. S. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 1994), ACM, pp. 290–301.

[4] BERGER, E. D., AND ZORN, B. G. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2006), ACM, pp. 158–168.

[5] BERGER, E. D., ZORN, B. G., AND MCKINLEY, K. S. Reconsidering custom memory allocation. *SIGPLAN Not. 37*, 11 (2002), 1–12.

[6] BOEHM, H.-J., AND WEISER, M. Garbage collection in an uncooperative environment. In *Software Practice & Experience* (New York, NY, USA, 1988), vol. 18, John Wiley & Sons, Inc., pp. 807–820.

[7] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium* (Berkeley, CA, USA, 2005), USENIX Association, pp. 177–192.

[8] DHURJATI, D., AND ADVE, V. Efficiently detecting all dangling pointer uses in production servers. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 269–280.

[9] DHURJATI, D., KOWSHIK, S., ADVE, V., AND LATTNER, C. Memory safety without runtime checks or garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES)* (2003), pp. 69–80.

[10] EVANS, J. A scalable concurrent malloc(3) implementation for FreeBSD. BSDCan, Apr. 2006.

[11] FENG, Y., AND BERGER, E. D. A locality-improving dynamic memory allocator. In *Proceedings of the Workshop on Memory System Performance (MSP)* (New York, NY, USA, 2005), ACM, pp. 68–77.

[12] GAY, D., ENNALS, R., AND BREWER, E. Safe manual memory management. In *Proceedings of the 6th International Symposium on Memory Management (ISMM)* (New York, NY, USA, 2007), ACM, pp. 2–14.

[13] GREENWALD, M., AND CHERITON, D. The synergy between non-blocking synchronization and operating system structure. *SIGOPS Oper. Syst. Rev. 30*, SI (1996), 123–136.

[14] GROSSMAN, D., MORRISETT, G., JIM, T., HICKS, M., WANG, Y., AND CHENEY, J. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2002), ACM, pp. 282–293.

[15] JONES, R. W. M., AND KELLY, P. H. J. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging (AADEBUG)* (1997), pp. 13–26.

[16] LVIN, V. B., NOVARK, G., BERGER, E. D., AND ZORN, B. G. Archipelago: trading address space for reliability and security. *SIGOPS Oper. Syst. Rev. 42*, 2 (2008), 115–124.

[17] MITRE CORPORATION. Common vulnerabilities and exposures (CVE). `http://cve.mitre.org`.

[18] MITRE CORPORATION. CWE-416: Use After Free. `http://cwe.mitre.org/data/definitions/416.html`.

[19] ROBERTSON, W., KRUEGEL, C., MUTZ, D., AND VALEUR, F. Run-time detection of heap-based overflows. In *Proceedings of the 17th USENIX Conference on System Administration (LISA)* (Berkeley, CA, USA, 2003), USENIX Association, pp. 51–60.

[20] SOLAR DESIGNER. “return-to-libc” attack. Bugtraq, Aug. 1997.

[21] SOTIROV, A. Heap feng shui in JavaScript. In *Black Hat Europe Briefings* (Feb. 2007).

[22] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC Benchmarks. `http://www.spec.org`.

[23] STRACKX, R., YOUNAN, Y., PHILIPPAERTS, P., PIESSENS, F., LACHMUND, S., AND WALTER, T. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security (EUROSEC)* (New York, NY, USA, 2009), ACM, pp. 1–8.

[24] XU, W., DUVARNEY, D. C., AND SEKAR, R. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE)* (New York, NY, USA, 2004), ACM, pp. 117–126.

[25] ZHAO, Q., RABBAH, R., AND WONG, W.-F. Dynamic memory optimization using pool allocation and prefetching. *SIGARCH Comput. Archit. News 33*, 5 (2005), 27–32.