Efficiently Detecting All Dangling Pointer Uses in Production Servers*

Dinakar Dhurjati Vikram Adve
University of Illinois at Urbana-Champaign
201 N. Goodwin Avenue
Urbana, IL - 61801, U.S.A.
{dhurjati, vadve}@cs.uiuc.edu

Abstract

In this paper, we propose a novel technique to detect all *dangling pointer uses at run-time that is efficient enough for production use in server codes. One idea (previously used by Electric Fence, PageHeap) is to use a new virtual page for each allocation of the program and rely on page protection mechanisms to check dangling pointer accesses. This naive approach has two limitations that makes it impractical to use in production software: increased physical memory usage and increased address space usage. We propose two key improvements that alleviate both these problems. First, we use a new virtual page for each allocation of the program but map it to the* same *physical page as the original allocator. This allows using nearly identical physical memory as the original program while still retaining the dangling pointer detection capability. We also show how to implement this idea* without *requiring any changes to the underlying memory allocator. Our second idea alleviates the problem of virtual address space exhaustion by using a previously developed compiler transformation called Automatic Pool Allocation to reuse many virtual pages. The transformation partitions the memory of the program based on their lifetimes and allows us to reuse virtual pages when portions of memory become inaccessible. Experimentally we find that the run-time overhead for five unix servers is less than 4%, for other unix utilities less than 15%. However, in case of allocation intensive benchmarks, we find our overheads are much worse (up to 11x slowdown).*

1 Introduction

Uses of pointers to freed memory ("dangling pointer errors") are an important class of memory errors responsible for poor reliability of systems written in C/C++ languages. These errors are often difficult and time consuming to find and diagnose during debugging. Furthermore, these dangling pointer errors can also be exploited in much the same way as buffer overruns to compromise system security [21]. In fact, many exploits that take advantage of a subclass of these errors (double free vulnerabilities) in server programs have been reported in bugtraq (e.g., CVS server double free exploit [7], MIT Kerberos 5 double free exploit [2], MySQL double free vulnerability [1]). Efficient detection of all such errors in servers during deployment (rather than just during development) is crucial for security.

Unfortunately, detecting dangling pointer errors in programs has proven to be an extremely difficult problem. Detecting such errors statically in any precise manner is undecidable. Detecting them efficiently at run-time while still allowing safe reuse of memory can be very expensive and we do not know of any practical solution that has overheads low enough for use in production code.

A number of approaches (including [3, 8, 9, 13, 16, 15, 17, 19, 20]) have been proposed that use some combination of static and run-time techniques to detect several kinds of memory errors, including buffer overflow errors and some dangling pointer errors. All of these techniques either have prohibitively high run-time overheads (2x - 100x) or memory overheads (or both) and are unsuitable for production software. Purify [8] and Valgrind [17], two of the most widely used tools for debugging memory access errors, often have overheads in excess of 1000% and can sometimes be too slow even for debugging long-running programs. Moreover, most of these approaches (except FisherPatil [15], Xu et al [19] and Electric Fence [16]) employ only heuristics to detect dangling pointer errors and do not provide any guarantees about absence of such errors. FisherPatil and Xu et al, detect all dangling pointer errors but perform software run-time checks on *all individual* loads and stores, incurring overheads up to 300% and also causing substantial increases in virtual and physical memory consumption (1.6x-4x). Electric Fence uses page protection mechanisms to detect all dangling pointer errors but does so at the expense of several fold increase in virtual *and* physical memory consumption of the applications.

*This work is supported in part by the NSF Embedded Systems program (award CCR-02-09202), the NSF Next Generation Software Program (award CNS 04-06351), and an NSF CAREER award (EIA-0093426).

1.1 Our approach

In this paper, we propose a new technique that can detect dangling pointers in server code with very low overheads, low enough that we believe they can be used in production code (though theyare also useful for debugging). Our approach builds on the naive idea (previously used in Electric Fence [16], PageHeap [13]) of using a a new virtual and physical page for each allocation of the program. Upon deallocation, we change the permissions on the individual virtual pages and rely on the memory management unit (MMU) to detect all dangling pointer accesses. This naive idea has two problems that make it impractical for any use other than debugging: increased address space usage (one virtual page for each allocation) and increased physical page usage (one page for each allocation). Our technique is based on two key insights that alleviates both these problems. Our first insight is based on the observation that even when using a new virtual page for each allocation we can still use the underlying physical page using a different virtual page that maps to that physical page. Our approach exploits this idea by using a new virtual page for each allocation of the program but mapping it to the same physical page as the original program (thus using the same amount of physical memory as the original program). Upon deallocation, we can change the permissions on the individual virtual pages but still use the underlying physical memory via different virtual pages. We rely on the memory management unit (MMU) just like in the naive idea to detect all dangling pointer accesses *without* any software checks. If the goal is to guarantee absence of undetected dangling pointer dereferences, then this basic scheme will not allow us to reuse a virtual page ever again for any other allocation in the program. Our second insight is that we can build on a previously developed compiler transformation called Automatic Pool Allocation [11] to alleviate the problem of address space exhaustion. The transformation essentially partitions the memory used by the program in to pools (sub heaps) and is able to infer when a partition or a pool is no longer accessible (using a standard compiler analysis known as escape analysis that is much simpler, but can be less precise, than that required for static detection of dangling pointer references). We leverage this information, to safely reuse address space belonging to a pool, when the memory corresponding to a pool becomes inaccessible.

As our experimental results indicate, our approach works extremely well for server programs. This is because most server programs seem to follow a simple memory allocation and usage paradigm: They have low or moderate frequency of allocations and deallocations but do have many memory accesses. Our approach fits well with this paradigm: we move all the run-time overheads to allocation and deallocation points (since we require extra system call per allocation and deallocation), and do not perform any checks on individual memory accesses themselves.

Our approach has several practical strengths. First, we do not use fat pointers or meta-data for individual pointers. Use of such meta-data complicates interfacing with existing libraries and requires significant effort to port programs to work with libraries. Second, if reuse of address space is not important [1], particularly during debugging, our technique can be directly applied on the binaries and does not require source code; we just need to intercept all calls to malloc and free from the program. Finally, we do not change the cache behavior of the program; so carefully memory-tuned applications can benefit from our approach without having to retune to a new memory management scheme.

There are two main limitations to our approach. First, since we use a system call on every memory allocation, applications that do in fact perform a lot of allocations and deallocations will have a big performance penalty (our approach can still be used for debugging such applications). However, we expect many security critical server software to not exhibit this behavior. Second, since each allocation has a new virtual page, our approach has more TLB ("translation lookaside buffer") misses than the original program. We are currently investigating simple architectural improvements that can mitigate both of these problems by changing the TLB structure.

We briefly summarize the contributions of this paper:

- We propose a new technique that can effectively detect *all* dangling pointer errors by making use of the MMU, while still using the same physical memory as the original program.

- We propose the use of previously developed compiler transformation called Automatic Pool Allocation to reduce the problem of address space exhaustion.

- We evaluate our approach on five unix utilities, five daemons and on an allocation intensive benchmark suite. Our overheads on unix utilities are less than 15% and on server applications are less than 4%. However, our overheads on allocation intensive benchmark suite are much worse (up to 11x slowdown).

The rest of this paper is organized as follows. The next section gives the necessary background for the rest of the paper. Section 3 contains a detailed description of our overall approach and our implementation. Section 4 gives experimental evaluation of our approach. Section 5 discusses related work and Section 6 concludes with possible future directions of this work.

[1] As shown later in Section 3.4, on 64-bit systems programs will run for at least 9 hours before running out of virtual pages

2 Background

2.1 Memory errors in C : Terminology

Using terminology from SafeC [3], memory errors in C programs can be classifieds into two different types: (1) Spatial memory errors and (2) Temporal memory errors

Spatial memory errors in C programs include array bounds violations (i.e., buffer overrun) errors, uninitialized pointer dereferences (causing an access to an invalid address), invalid type conversion errors, format string errors, etc. Temporal memory errors include uses of pointers to freed heap memory and uses of pointers to an activation record after the function invocation completes.

In this work, we focus on detecting uses of pointers to freed heap memory. In previous work, we have described techniques for detecting spatial errors with very low overhead, which also exploits Automatic Pool Allocation to reduce run-time overhead [4]. Those techniques (and other approaches that detect spatial errors) are complementary to our approach here because our approach here does not use any metadata on individual pointers or objects and does not restrict adding such metadata. For dangling pointer accesses to stack objects, some combination of compile-time escape analysis, run-time checks, or converting possibly escaping stack allocations to heap allocations can be used [5, 14]. In the rest of this paper, by dangling pointer errors we mean use of pointers to freed heap memory, where use of a pointer is a read, write or free operation on that pointer.

2.2 Background on Automatic Pool Allocation

Given a program containing explicit `malloc` and `free` operations, Automatic Pool Allocation transforms the program to segregate data into distinct pools on the heap [11]. It uses the points-to graph of the program (essentially a static partitioning of the heap and stack objects in the program and the connectivity of these objects) to perform the segregation. Each node in the points-to graph represents a set of memory objects of the original program. Pool allocation creates a distinct pool, represented by a pool descriptor variable, for each points-to graph node that represents heap objects.

We explain the pool allocation transformation with the help of a simple example shown in Figure 1 [2]. In this example, function `f` calls `g`, which first creates a linked list of 10 nodes, initializes them, and then calls `h` to do some computation. `g` then frees all of the nodes except the head and then returns. Note that the program has a dangling pointer

[2]The same example was used in our previous work [6], but that work did not detect dangling dereferences

```
f() {
  struct s *p = 0;
  // p is local
  g(p);
  p->next->val =...; // p->next is dangling
}
g(struct s *p) {
  p->next = malloc(sizeof(struct s));
  create_10_Node_List(p);
  initialize(p);
  h(p);
  free_all_but_head(p);
}
```

Figure 1. Running example

error: the reference to `p->next->val` tries to access the second node in the list, which has been freed.

Figure 2 shows the running example after the Pool Allocation transformation. The transformation first identifies points-to graph nodes that do not "escape" a function using a traditional escape analysis (reachability analysis from function arguments, globals and return values) and creates pools for those nodes at the function entry and destroys them at the function exit. In the running example, the data structure pointed to by `p` never escapes the function `f()`, so the transformation inserts code to create a pool `PP` within `f` using `poolinit` and destroys at the function exit using `pooldestroy`. For a function where the pool "escapes" (e.g, function `g` in the running example) the transformation automatically modifies the function to take in extra pool descriptor arguments (see function `g` in Figure 2). Pool allocation then ensures that all allocations and deallocations for the data structure happen out of the corresponding pool – it converts `malloc` and `free` calls to use `poolalloc` and `poolfree` with the appropriate pool descriptors in the program. This is illustrated by the change of `malloc` call in function in `g` to `poolalloc`. Similarly `malloc` calls in `create_10_Node_list()` and `free` calls in `free_all_but_head` (not shown here) are also changed appropriately to use the corresponding pool calls. Finally all function invocations are modified to pass in the extra pool descriptor arguments (see invocations of `g()`, `create_10_Node_list()`, `free_all_but_head()` in Figure 2).

Note that explicit deallocation via `poolfree` can return freed memory to its pool and then back to the system, which can then allocate it to a different pool. Thus dangling pointers to the freed memory in the original program continue to exist in the transformed program.

In our previous work, we have used Automatic Pool Allocation to improve memory hierarchy performance [11] and to enforce memory safety [6] and sound alias analysis [5]. None of these detected dangling pointers. The cur-

```
f() {
  Pool PP;
  poolinit(&PP, sizeof(struct s));
  g(p, PP);
  p->next->val = ... ; //p->next is dangling
  pooldestroy(PP);
}

g(struct s *p, Pool *PP) {
  p->next = poolalloc(PP, sizeof(struct s));
  create_10_Node_List(p, PP);
  initialize(p);
  h(p);
  free_all_but_head(p, PP);
}
```

Figure 2. Example after pool allocation transformation

rent work is the first to consider how automatic pool allocation can be used to detect dangling pointer references.

3 Our Approach

3.1 Overview of the Approach

The memory management unit (MMU) in most modern processors performs a run-time check on every memory access (by looking at the permission bits of each page). Debugging tools like Electric Fence [16] and PageHeap [13] exploit this check to detect dangling pointer accesses at runtime. Since the MMU does checks only at page level, these tools allocate only one object per (virtual and physical) page and change the permissions at a free operation to protect the page. Any dangling pointer access will then result in a hardware trap, which can be caught. However, allocating only one object per *physical* page would quickly exhaust physical memory. Further more, changing the allocation pattern this way would potentially lead to poor cache performance in physically indexed caches. Our first insight addresses this drawback:

Insight1: *Mapping multiple virtual pages to the same physical page enables us to set the permissions on each individual virtual page separately while still allowing use and reuse of the entire physical page via different virtual pages.*

With this approach, the consumption of physical memory would be (nearly) identical to the original program, and the multiple objects could be contiguous within the page, preserving spatial locality in physically indexed caches. Moreover, as we show later, with a minor increase in memory allocation (one word per allocation) this scheme can be implemented *without* requiring any changes to the underlying memory allocator. In a practical system, this is likely to be significant, since programs that are already tuned to an existing memory allocation strategy do not need to be retuned again.

Virtual memory pages, however, still cannot be reused (to ensure that any access to a previously freed object is detected arbitrarily far in the future). As noted previously in Section 2, the Automatic Pool Allocation transformation provides bounds on the lifetimes of pools containing heap objects, such that *there are no live pointers to a pool* after the `pooldestroy` operation on the pool. This yields:

Insight2: *For a program transformed using Automatic Pool Allocation, it is safe to release all the virtual pages assigned to a pool at the `pooldestroy` operation on the pool.*

The main remaining limitation of this approach is that long lived pools (e.g., pools reachable via a global pointer, or pools created and destroyed in the `main` function for other reasons) effectively live throughout the execution and their virtual pages cannot be reused. In Section 3.4, we discuss three strategies to avoid this problem in practice.

Overall, we now have the ability to reuse physical pages as well as the original program does and reuse virtual memory pages partially. Nevertheless, there are several key implementation and performance issues that must be considered to make this approach practical for realistic production software. These issues are addressed as we describe the details of the approach below.

3.2 Page mapping for detecting dangling pointer errors

The primary mechanism we use for detecting dangling pointer references (i.e., a load, store, or free to a previously freed object) is to use a distinct virtual page or sequence of pages for every dynamically allocated object. When an object is freed, the protected bit is set for the page or pages using the `mprotect` system call so that any subsequent reference to the page causes an access violation, which is handled by our run-time system.

We assume in this subsection that a standard heap allocator is used. A useful property of our basic approach is that it can work with an arbitrary memory allocator and furthermore, requires only a small addition to the metadata for the allocator and no change to the allocation algorithm itself. In fact, *the underlying allocator is completely unaware of the page remapping used to ensure unique virtual pages.* The basic approach works as follows, assuming `malloc` and `free` are the interface for the underlying system allocator.

Allocation: An allocation request is passed to `malloc` with the size incremented by `sizeof(addr_t)` bytes; the extra bytes at the start of the object will be used to record an address for bookkeeping purposes. Let a be the address returned by `malloc`, $\texttt{Page}(a) = a \;\&\; \sim (2^p - 1)$ and $\texttt{Offset}(a) = a \;\&\; (2^p - 1)$, where p is log_2 of the VM page size. The latter two denote the start address of the page containing a and the offset of a relative to the

start of the page. We then invoke a system call to assign a fresh virtual page (or pages) that share the same physical memory as the page(s) containing a. On Linux, we do this using `mremap(old_address, old_size, new_size, flags)`, with `old_size = 0`, which returns a new page-aligned block of virtual memory at least as large as `new_size`.[3] If the new page address is P_{new}, we return $P_{new}(a) + \texttt{Offset}(a) + \texttt{sizeof(addr\_t)}$ to the caller.

Note that the underlying allocator still believes that the allocated object was at address a, whereas the caller sees the object on a different page but at the same location within the page. We refer to virtual page $\texttt{Page}(a)$ as the *canonical page* (the one assumed by the allocator) and the actual virtual page P_{new} as the *shadow page* for the object. We have to record the original page $\texttt{Page}(a)$ for the object to support deallocation; we do this in the extra `sizeof(addr_t)` bytes we allocated above. Note that malloc implementations usually add a header recording the size of the object just before the object itself so we are effectively extending that header to also record the value of $\texttt{Page}(a)$.

The net result of this approach is that multiple objects can live in a single physical page and the underlying allocator believes they live in a single virtual page (the canonical page). The program, however, is given a distinct virtual page (a shadow page) for each object in the physical page.

Deallocation: On a deallocation request for address f, we first look up the canonical page for this object which was recorded at $f - \texttt{sizeof(addr\_t)}$. Note that this read operation will cause a run-time error if the object has already been freed because the virtual page containing this memory will have been protected as explained next. We read the size of the object using the metadata recorded by `malloc`, use this size to compute how many pages the object spanned, and use the system call `mprotect` to set the protection status of the page(s) containing f to the state `PROT_NONE`. This will cause any future read or write of the page to trap. If the canonical page is $\texttt{Page}_C^f$, we invoke $\texttt{free(Page}_C^f + \texttt{Offset}(f)\texttt{)}$ to free the object. The allocator marks the object within the canonical page as free, allowing that range of virtual memory (and therefore the underlying physical memory) to be reused for future allocations. The shadow page(s) containing the object at f cannot be reused, at least with the approach as described so far.

To illustrate the primary mechanism described so far, consider the dangling pointer example in Figure 1. With our page remapping scheme, each list node will be allocated to a fresh virtual page but will share the underlying physical pages with other nodes. At the end of `g`, all pages except the one for the head node will be marked protected, so that the dangling pointer reference will cause a trap. The physical memory used by this code will be identical to the original program, except for the small extra metadata on each live object that has not been freed. Unfortunately, every call to `f()` will construct a new list, consuming new virtual pages each time and these virtual pages will not be reused. The next technique, however, solves this problem completely for this example.

[3]This behavior is undocumented in the man page but is described here [18]. On systems where this feature is not available, we can use `mmap` with an in-memory file system.

3.3 Reusing virtual pages via Automatic Pool Allocation

In Automatic Pool Allocation, each pool created by the program at run-time is essentially a distinct heap, managed internally using some allocation algorithm. We can use the remapping approach described above *within each pool* created by the program at run-time. The key benefit is that, at a `pool_destroy`, we can release all (shadow and canonical) virtual memory pages of the pool to be reused by future allocations. Note that physical pages will continue to be reused just as in the original program, i.e., the physical memory consumption remains the same as in the original program (except for minor differences potentially caused by using the pool allocator on top of the original heap [11]).

The only significant change in the Allocation and Deallocation operations described above is for reusing virtual pages. This is slightly tricky because we need to reuse virtual pages that might have been aliased with other virtual pages previously. One simple solution would be to use the `unmap` system call to release previous virtual-to-physical mappings for all pages in a pool after a pool destroy. `unmap` would work for both canonical and shadow pages because these are obtained from the Operating System (OS) via *mmap* and *mremap* respectively. Canonical pages are obtained in contiguous blocks from the underlying system (via *mmap*) and the blocks can be unmapped efficiently. The shadow pages, however, are potentially scattered around in the heap, and in the worst case may require a separate `unmap` operation for every individual object allocated from the pool (in addition to the earlier `mprotect` call when the object was freed). This could be expensive.

We avoid the explicit `munmap` calls by maintaining a free list of virtual pages shared across pools and adding all pool pages to this free list at a `pool_destroy`. We modified the underlying pool allocator to obtain (canonical) pages from this free list, if available. If this free list is empty, we use *mmap* to obtain fresh pages from the system as before. For each allocated object, the shadow page(s) is(are) obtained using *mremap* as before to ensure a distinct virtual memory page.

A `pool_free` operation works just like the Deallocation case described previously, and invokes the underlying `pool_free` on the canonical page. A `pool_destroy`

operation simply returns all canonical *and* shadow pages in the pool to the shared free list of pages.

Considering the example again but after pool allocation (Figure 2), a fresh pool is created for each list on entry to `f()` and destroyed before returning from `f()`. Within this pool, nodes are allocated on separate pages, the pages are protected on free, and the dangling pointer reference is detected as before. On return from `f()`, however, all the virtual pages of the pool will be released to the free list and reused for future allocations (in future invocations of `f()` or elsewhere). The key property is that the compiler was able to prove that no pointers to the pool are reachable after `f()` returns. This is much simpler than compile-time detection of dangling pointers, which would have required predicting at the reference to `p->next->val` that specific objects within the list have been freed.

3.4 Avoiding costs of long-lived pools

As noted earlier in Section 3.1, the main limitation of our approach as described so far is that virtual pages in pools with lifetimes spanning the entire execution will never be reused. Our examination of several Unix daemons in Section 4.3 shows that this problem arises in very few cases.

If it occurs in a specific program, it would impose two costs in practice: (1) A long-running program may eventually run out of virtual memory. (2) Small operating system resources (the page table entry) are tied up for each non-reusable virtual page. The page cannot be unmapped to prevent reuse of the virtual addresses. We propose three solutions to avoid these problems in practice.

The simplest solution is to start reusing virtual pages when we run out of virtual addresses, or at some regular (but large) interval. A simple calculation shows that on a 64-bit Linux system (and assuming a maximum of 2^{47} bytes of virtual memory for a user program), even an extreme program that allocates a new 4K-page-size object *every microsecond*, with no reuse of these pages, can operate for 9 hours before running out of virtual pages ($2^{47}/(2^{12} \times 10^6 \times 86,400)$). In practice, even with no reuse at all, we expect typical servers to be able to operate for days before running out. The small probability of not detecting a dangling pointer in such situations appears unimportant. In practice, therefore, the second cost above (tying up OS resources) appears to pose a tighter constraint than the first. For this reason, real-world applications would likely choose to reuse memory after a shorter (but still infrequent) interval.

An alternative approach is to run a conservative garbage collector (GC) at the same infrequent intervals (based on the same criteria above) to release the tied-up virtual addresses. This is much simpler and less expensive than using garbage-collection for overall memory management for two reasons. Most importantly, since the actual physical memory consumption is not an issue and GC only needs to ameliorate the two problems above, we can run garbage collection quite infrequently (e.g., once every few hours) and when there is a light load on the server. Second, we only need to use GC to collect memory from the long-lived pools (which are known to the pool allocation transformation). We already have a very simple facility in our system for each run-time pool descriptor to record exactly which currently live pools point to it, i.e., a "dynamic pool points-to graph" [12]. By knowing which pools need to be collected, the collector can use this information to traverse only a subset of the heap.

If the first solution is not acceptable for some reason, and conservative GC is not available or unattractive, a third alternative is that the programmer could modify the application so that fewer heap objects are reachable from global pointers. This is similar to, but a subset of, the tuning needed to reduce memory consumption of applications that use GC for memory management, since the latter also requires resetting local and global pointer variables to null as data structures are freed.

Overall, we believe that with one or more of these techniques, the potential costs of lack of reuse in globally live pools is likely to be unimportant in real-world systems.

3.5 Implementation

We have implemented the techniques described in this paper in the pool allocation run-time system developed over the course of our previous work on performance optimization [11] and on memory safety [6, 5, 4].

We use the LLVM compiler infrastructure [10] to apply Automatic Pool Allocation to C programs, using the existing version of this transformation with no changes. We modified the pool allocator run-time in minor ways to implement the techniques described earlier. We modified `pooldestroy` to return all pages to a shared free list of pages. We also modified `poolfree` so it did not return unused blocks to this free list. We modified `poolalloc` to try to obtain fresh pages from the free list first when it needs fresh pages (and falling back on `mmap` as before, if the free list is empty). Finally, we wrap the calls to `poolalloc` and `poolfree` to remap objects from canonical to shadow pages and vice-versa, as described earlier.

4 Experimental Evaluation

We present an experimental evaluation of our approach for unix utilities, server applications and some allocation intensive applications. The goal of these experiments is to measure the net run-time overhead of our approach, a breakdown of these overheads contributed by different factors,

Benchmark	LOC	Execution time in Secs					Slowdown ratios	
		native	LLVM (base)	PA	PA + dummy syscalls	Our approach	Ratio 1	Ratio 2
Utilities								
enscript	8514	1.135	1.077	1.177	1.143	1.238	**1.15**	1.09
jwhois	10702	0.539	0.534	0.539	0.535	0.534	**1.00**	0.99
patch	11669	0.174	0.176	0.177	0.179	0.179	**1.02**	1.03
gzip	8163	4.509	5.419	5.01	4.91	4.943	**0.91**	1.10
Servers								
ghttpd	6036	4.385	4.507	4.398	4.461	4.486	**1.00**	1.02
ftpd	23033	2.236	2.293	2.267	2.318	2.291	**1.00**	1.02
fingerd	1733	1.238	1.285	1.277	1.278	1.285	**1.00**	1.04
tftpd	880	2.246	2.281	2.289	2.293	2.287	**1.00**	1.02

Table 1. Runtime overheads of our approach. Ratio 1 is the ratio of execution time of our approach with respect to base LLVM, Ratio 2 is the ratio of execution time of our approach to native code. (Two other applications, *telnetd* and *less*, are discussed in text)

and the potential for unbounded growth in the virtual address space usage incurred in our approach. Note that our physical memory consumption is almost exactly the same as the original program (except for minor differences when using the pool allocation runtime library) and we do not evaluate the physical memory consumption experimentally.

4.1 Run-time overheads for system software

We evaluated our approach using some commonly used unix utilities and five server codes – ghttpd-1.4, wu-ftpd-2.6 , bsd-finged-0.17, netkit-telnet-0.17, netkit-tftp-0.17. The characteristics of these applications are listed in Table 1. We compiled each program to the LLVM compiler intermediate representation (IR), perform our analyses and transformations, then compile LLVM IR back to C and compile the resulting code using GCC 3.4.2 at -O3 level of optimization. We performed our experiments on a (32-bit) Intel Xeon with Linux as the operating system. For each of the server applications, we generated a list of client requests and measured the response time for the requests. We ran the server and the client on the same machine to eliminate network overhead. We conducted each experiment five times and report the median of the measured times. In case of utilities, we chose a large input size to get reliable measurements. We successfully applied our approach to two interactive applications `netkit-telnetd` and unix utility `less` and did not notice any perceptible difference in the response time. We do not report detailed timings for these two applications.

The "native" and "LLVM (base)" columns in the table represent execution times when compiled directly with GCC -O3 and with the base LLVM compiler (without pool allocation or any of our mmap system calls) using the LLVM C back-end. LLVM uses a different set of optimizations than GCC so there is a (minor) difference in the two execution times. Using LLVM (base) times as our baseline allows us to isolate the affect of the overheads added by our approach. The "native" column shows that the LLVM (base) code quality is comparable to GCC and reasonable enough to use as a baseline. The "PA" column shows the time when we only run the pool allocator and do not use our virtual memory technique, i.e., it shows the effect of pool allocation alone on execution time.

As noted earlier in Section 1, overheads in our approach could be due to two reasons: (1) use of a system call on every allocation (`mremap`) and deallocation (`mprotect`) (2) TLB miss penalty because we use far more virtual pages than the original program. The "PA + dummy syscalls" column shows the execution time when we do a dummy `mremap` system call on every allocation and a dummy `mprotect` system call on deallocation. This allows us to isolate the overheads due to system calls from that of TLB misses. The "Our approach" column gives the total execution time with our approach.

Ratio 1 gives the ratio of execution time of our approach to that of LLVM (base). Ratio 2 gives the ratio of execution time of our approach to that of native code.

As we can see from column **Ratio1**, our overheads are negligible for most applications. Only one application, `enscript`, has a 15% overhead. These overheads are much better than the any one of the previous approaches for detecting dangling pointers [16, 19, 15]. `enscript` does many allocations and deallocations (in fact, when used with electric fence, `enscript` runs out of physical memory). From the "PA + dummy syscalls" column, we can see that the overhead in `enscript` due to system calls is about 6%. We attribute the remaining component of the overhead (around 9%) to TLB miss penalty. Automatic Pool Allocation transformation can sometimes speedup applications

Benchmark	Execution time (Secs)		Slowdown ratios	
	Our approach	Valgrind	Our slowdown	Valgrind slowdown
enscript	1.238	29.931	**1.15**	**26.37**
jwhois	0.534	1.336	**1.00**	**2.48**
patch	0.179	1.461	**1.02**	**8.40**
gzip	4.943	94.483	**0.91**	**20.95**

Table 2. Comparison with Valgrind. Our slowdown ratio is Ratio 1 from Table 1

.

(e.g., `gzip`) because of better cache performance [11].

Overall, we believe our low overheads are due to the patterns of memory allocation and use that servers seem to obey: there are relatively few allocations/deallocations (keeping our system call overheads low) but potentially many uses of the allocated memory (we do not incur overheads due to hardware checks).

4.2 Comparison with Valgrind

We compared the overheads of our approach with Valgrind [17] (a widely used open source debugging tool) on the four Unix utilities. The servers are spawned off the `xinetd` process for every client request and we were unable to run them under Valgrind. The results are shown in Table 2. Valgrind attempts to detect both spatial errors and some dangling pointer errors. We cannot isolate Valgrind overhead for temporal errors, so the comparison is only meant to give a rough indication of the magnitude of difference in overhead. It is worth noting that Valgrind uses a heuristic to detect dangling pointer errors (see Section 5 for more discussion) and does not guarantee detection of dangling pointer accesses. The overheads for Valgrind range from 148% to 2537%, which is orders-of-magnitude worse than ours. In contrast, our approach detects only dangling pointer errors, but our overheads are negligible for three of the applications and 15% for one application. In our previous work [4, 5], we detect all memory errors *except* dangling pointer errors and if those techniques were combined with ours, our cumulative overheads would still be much lower than that of Valgrind for these applications.

4.3 Address space wastage due to long-lived pools

We studied the usage (and wastage) of virtual address space incurred by our technique by tracing three of the server programs (`ftpd`, `telnetd`, `ghttpd`) using gdb. We focused on the servers since they are long-running (as well as security-critical) programs. We found that a common programming model used by these servers is to fork a new process to service each new connection. Although we did not trace fingerd and tftpd, it is clear from the comments in the source code that they follow exactly the same programming model; in fact, in case of tftpd every command from the client (e.g, get filename) forks off a new process. This model of forking a new process to service requests fits well with our approach. Any wastage in address space in one connection is not carried over to the other connections handled by the server. We expect each individual connection to be of short duration even though all the servers themselves are long running.

We then measured the usage of virtual address space within each individual connection for the three servers.

`ghttpd` is a webserver designed for small memory foot print and performs only one dynamic allocation per connection. Consequently, there is no virtual memory wastage when we use our approach.

In case of `ftpd`, we found that in a few cases, the pool allocation transformation helps in reuse of address space. For example, the `fb_real_path` function in `ftpd`, which resolves sym links, first creates a pool, allocates some memory out of the pool, does some computation, frees the memory, and finally destroys the pool. Any virtual addresses used by this pool are reusable after the pool destroy. However, there are other allocations in `ftpd` that are out of global pools and do not benefit from pool allocation. We found that for each `ftp` command (e.g., get filename), there are 5-6 allocations from global pools, so that virtual memory usage increases at the rate of 5-6 pages per command. Although this problem could be alleviated using the techniques described in Section 3.4, this problem is unlikely to be important for `ftpd` because the process is killed at end of a user connection.

`telnetd` performs 45 small allocations (and deallocations) before giving control to the shell in each session (process). It does not do any more (de)allocations and just waits for the session to end. Using our approach we just use 45 virtual pages for each session. In all these cases, we *guarantee* the absence of any undetected dangling pointer accesses.

4.4 Overheads for allocation intensive applications

We measured the run-time overheads of our approach when applied to allocation intensive Olden benchmarks. These benchmarks have high frequency of allocations and are a worst case scenario for our approach. While the overheads for three of the Olden benchmarks were less than 25%, the overheads for the remaining six are high (slowdowns from 3.22 to 11.24). As can be noted from several programs, including `bisort`, `health`, and `mst`), the overheads can be attributed to both the system call overheads and TLB misses. Our approach can be used

	native	LLVM (base)	PA + dummy sys call	Our approach	Ratio 3
bh	15.090	9.723	11.035	12.127	**1.25**
bisort	2.803	2.641	4.740	8.495	**3.22**
em3d	9.774	6.830	7.366	7.801	**1.14**
health	0.319	0.305	2.355	3.429	**11.24**
mst	0.285	0.166	1.040	1.582	**9.53**
perimeter	0.187	0.210	1.428	2.188	**10.42**
power	5.698	2.903	2.959	3.168	**1.09**
treeadd	0.277	0.293	0.455	1.0777	**3.68**
tsp	1.753	1.637	3.647	6.749	**4.12**

Table 3. Overheads for allocation intensive Olden benchmarks, Ratio 3 is the slow down of our approach with respect to LLVM base.

for such allocation (and deallocation) intensive applications only during debugging.

5 Related Work

Detecting memory errors in C/C++ programs is a well researched area. A number of previous techniques focus only on spatial memory errors (buffer overflow errors, uninitialized pointer uses, arbitrary type cast errors, etc). As explained in Section 2.1, detecting spatial errors is complementary to our approach. In this section, we compare our approach to only to those techniques that detect or eliminate dangling pointer errors.

One way to eliminate dangling pointer errors is to use automatic memory management (garbage collection) instead of explicit memory allocation and deallocation. Where appropriate, that solution is simple and complete, but it can significantly impact the memory consumption of the program and perhaps lead to unacceptable pause times. For C and C++ programs that choose to retain explicit frees in the program, an alternative solution is required. We focus here on comparing those approaches that detect dangling pointers in the presence of explicit deallocation.

5.1 Techniques that rely on heuristics to detect dangling pointer bugs

A number of systems have been proposed that use heuristic run-time techniques to detect heap errors, including some dangling pointer errors [20, 17, 9, 8]. As noted in the Introduction, these techniques do not provide any guarantees about the absence of such errors. The limitation is indeed significant: in fact, these techniques can detect dangling memory errors only as long as the freed memory is not reused for other allocations in the program. Furthermore, all of them rely on heuristics to delay reuse of freed memory, which can increase the *physical* memory consumption.

5.2 Techniques that guarantee absence of dangling pointer references

SafeC [3] is one of the earliest systems to detect (with high probability) all memory errors including all dangling pointer errors in C programs. SafeC creates a unique capability (a 32-bit value) for each memory allocation and puts it in a Global Capability Store (GCS). It also stores this capability with the meta-data of the returned pointer. This meta-data gets propagated with pointer copying, arithmetic. Before every access via a pointer, its capability is checked for membership in the global capability store. A free removes the capability from the global capability store and all dangling pointer accesses are detected. FisherPatil [15] and Xu et. al [19] propose improvements to the basic scheme by eliminating the need for fat-pointers and storing the meta-data separately from the pointer for better backwards compatibility. To be able to track meta-data they disallow arbitrary casts in the program, including casts from pointers to integers and back. Their overheads for detecting only the temporal errors on allocation intensive Olden benchmarks are much less than ours – about 56% on average (they do not report overheads for system software).

However, the GCS can consume significant memory: they report increases in (physical and virtual) memory consumption of factors of 1.6x - 4x for different benchmarks sets [19]). For servers in particular, we believe that such significant increases in memory consumption would be a serious limitation.

Our approach, on the other hand, provides better backwards compatibility: we allow arbitrary casts including casts from pointers to integers and back. Furthermore, our approach uses the memory management unit to do a hardware runtime check and does not incur any per access penalty. Our overheads in our experiments on system software, with low allocation frequency, are negligible in most cases and less than 15% in all the cases. However, for programs that perform frequent memory allocations and deallocations like the Olden benchmarks, our overheads are significantly worse (up to 11x slowdown). It would be an interesting experiment to see if a combination of these two techniques can work better for general programs.

5.3 Techniques that check using MMU

As mentioned earlier, Electric Fence [16] and PageHeap [13] are debugging tools that make use of the memory management unit (MMU) to detect dangling pointer errors (and some buffer overflow errors) without inserting any software checks on individual loads/stores. However, both

the tools allocate only one memory object per virtual and physical page, and do not attempt to share a physical page through different virtual pages. This means that even small allocations use up a page of actual physical memory. This results in several fold increase in memory consumption of the applications. In turn, the applications exhibit very bad cache behavior increasing the overheads of the tools. These overheads effectively limit the usefulness of the tools to debugging environments. In contrast, we share and reuse physical memory, and use automatic pool allocation to reuse virtual addresses.

6 Conclusion and Future Work

In this paper, we have presented a novel technique to detect uses of pointers to freed memory that relies on two simple insights: (1) Using a new virtual page for every allocation but mapping it to the same physical page as the original allocator. (2) Using a compiler transformation called automatic pool allocation to mitigate the problem of virtual address space exhaustion. We evaluated our approach on several Unix utilities and servers and we showed that our approach incurs very low overhead for all these cases – less than 4% for the server codes and less than 15% for the utilities. *These overheads are low enough to be acceptable even in production code* (although our techniques could be very effective for debugging as well). We believe this is the first time such a result has been demonstrated for the run-time detection of dangling pointer errors.

For C or C++ programs that have frequent allocations and deallocations, two main performance problems remain — the system call overhead for allocations and deallocations, and the TLB miss overhead. As an extension to this work, we plan to investigate simple OS and architectural enhancements that can reduce both these kinds of overheads and make our approach applicable to these other kinds of software. We also plan to combine this approach with techniques for detecting other kinds of memory errors from our previous work [4, 5], to build a comprehensive safety checking tool. We believe this will be straightforward as the compiler and run-time techniques are complementary, and have been implemented in a common infrastructure.

References

[1] MySQL double free heap corruption vulnerability. `http://www.securityfocus.com/bid/6718/info`, Jan 2003.

[2] MITKRB5-SA: double free vulnerabilities. `http://seclists.org/lists/bugtraq/2004/sep/0015.html`, Aug 2004.

[3] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 1994.

[4] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proc. 28th Int'l Conf. on Software Engineering (ICSE)*, Shanghai, China, May 2006.

[5] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2006.

[6] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without garbage collection for embedded applications. *ACM Transactions on Embedded Computing Systems*, Feb. 2005.

[7] I. Dobrovitski. Exploit for cvs double free() for linux pserver. `http://seclists.org/lists/bugtraq/2003/feb/0042.html`, Feb 2003.

[8] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter USENIX*, 1992.

[9] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[10] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, San Jose, Mar 2004.

[11] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun 2005.

[12] C. Lattner and V. Adve. Transparent Pointer Compression for Linked Data Structures. In *MSP*, Chicago, IL, Jun 2005.

[13] Microsoft. How to use Pageheap.exe in Windows XP and Windows 2000. `http://support.microsoft.com/?kbid=286470` .

[14] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Language and Systems*, 27(3):477–526, 2005.

[15] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Software–Practice and Experience*, 27(1):87–110, 1997.

[16] B. Perens. Electric fence *malloc* debugger. `http://perens.com/FreeSoftware/ElectricFence/` .

[17] J. Seward. Valgrind, an open-source memory debugger for x86-gnu/linux.

[18] L. Torvalds. mremap feature discussion, see `http://lkml.org/lkml/2004/1/12/265`.

[19] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proc. 12th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 117–126, 2004.

[20] S. H. Yong and S. Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Foundations of Software Engineering*, 2003.

[21] Y. Younan. An overview of common programming security vulnerabilities and possible solutions. Master's thesis, Vrije Universiteit Brussel, 2003.