HeapShield: Library-Based Heap Overflow Protection for Free

Emery D. Berger

Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
emery@cs.umass.edu

Abstract

While numerous approaches have been proposed to prevent stack overflows, heap overflows remain both a security vulnerability and a frequent source of bugs. Previous approaches to preventing these overflows require source code or can slow programs down by a factor of two or more. We present HeapShield, an approach that prevents all library-based heap overflows at runtime. It works with arbitrary, unaltered binaries. It incurs no space overhead and is efficient, imposing minimal impact on real application performance (2% on average).

1. Introduction

The C library contains a host of library calls that present well-known security vulnerabilities. The most prominent culprit is `strcpy`, which overflows the destination buffer if the source string is too large. Numerous other examples exist, including `gets`, `sprintf`, and `memcpy`.

Because of the danger inherent in the use of these unsafe library calls, a number of safer alternatives have been added to the C library. These variants typically include the letter "n" in their name (e.g., `strncpy`) and require an argument that gives the maximum amount of data copied into the target buffer. Unfortunately, these variants do little to prevent overflows, because programmers can simply supply incorrect values.

Preventing buffer overflows has been the object of substantial recent research efforts, especially with the goal of preventing stack-smashing attacks. However, buffer overflows can also be used to attack the heap, and approaches suitable for preventing stack-smashing, such as canaries or shadow stacks, are ineffective against heap overflows. The current state-of-the-art approach prevents heap overflows but degrades performance by up to 2.4X.

This paper presents a memory-management based approach that efficiently prevents all library-based heap overflows. The key insight we exploit in this paper is that it is possible to employ an alternative heap layout to track object sizes "for free". This heap layout, described in detail in Section 3.1, differs substantially from that used by the Linux and Windows allocators. It has been used in the programming languages community to provide type information for LISP objects with low overhead [15], to support conservative garbage collection [11], and as the foundation for scalable multiprocessor memory management [6, 21]. It has also been used as the basis for the default memory manager in the FreeBSD operating system [19].

We show here how we can apply this heap layout to ensure that library functions never cause heap overflows. Our approach incurs no space overhead beyond that used to manage the heap because it relies only on data structures already in place to support ordinary memory allocation operations (`malloc` and `free`). We use this heap layout to efficiently implement a function that computes the remaining size of any allocated buffer, and then replace all unsafe library functions with safe variants that silently truncate overflows.

HeapShield imposes no perceivable performance impact on these library functions, and it adds no overhead to any other aspect of program execution. It is also orthogonal to techniques that protect against stack overflow and so could be combined with them, although we do not explore that here.

The remainder of this paper is organized as follows. Section 2 explains the difficulties of efficiently preventing heap overflows at the language level. Section 3 describes our approach in detail. Section 4 presents experimental results across microbenchmarks and actual applications. Finally, Section 5 discusses related work and Section 6 concludes.

2. Overview

Library calls like `strcpy` are unsafe because they do not check whether copying the source string would overflow the destination buffer. The reason that standard library implementations do not perform such checking is that it can be prohibitively expensive to do so, as we discuss below.

In particular, given any pointer, we need to be able to obtain the following information in order to prevent overflows:

1. Determine whether the pointer lies in the heap.
2. Find the position of this pointer inside an allocated object.
3. Compute the available space remaining from this pointer to the end of its allocated space.

Unfortunately, popular memory managers do not provide any ready way to obtain this information. The memory managers used by both Windows and Linux are *freelist-based*: they manage freed space on linked lists, and return pointers to the allocated objects. We focus here on the Lea allocator [20], which forms the basis of the Linux (GNU libc) allocator and is representative of freelist-based memory allocators.

As Figure 1 shows, the Lea allocator adds a header to each allocated object that contains its size and the size of the previous object. This metadata allows the Lea allocator to efficiently place freed objects on the appropriate free list (organized by size), and to *coalesce* adjacent freed objects into a larger chunk.

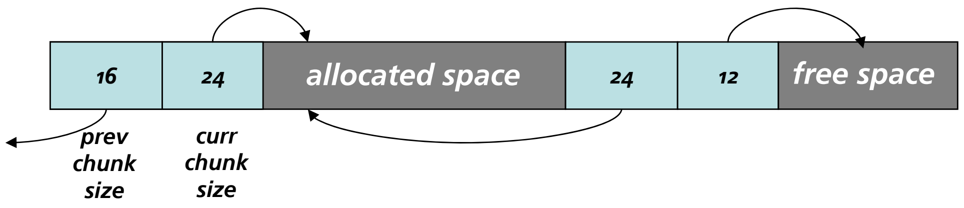


Figure 1. A fragment of a freelist-based heap, as used by Linux and Windows. Object headers precede each object, making it easy to free and coalesce objects, but provide no way to locate objects given an interior pointer.

However, this heap layout makes it impossible to efficiently or reliably locate objects given an interior pointer. One could perform a linear scan of memory searching for the metadata, which could prove quite costly. Even then, there is no sure way to distinguish between object metadata and the contents of the object itself.

Given these difficulties, the only way to track this information is to build an auxiliary data structure. Hauswirth and Chilimbi's *address tree* [17] is one recent example. This binary tree, despite its relative space efficiency versus na ive approaches, imposes both space costs (as much as 47%) and substantial performance overhead, requiring up to 32 tree traversals for each lookup. Similarly, Avijit et al.'s libsafePlus [2] uses a red-black tree to accomplish the same objectives, but this increases the cost of memory allocation operations by up to 7X, degrading application performance by up to 2.4X on one application.

In general, any auxiliary approach imposes a performance penalty on `malloc` and `free` operations (which are frequent), rather than placing overhead on far less frequent calls to functions like `strcpy`. Also, in the context of multi-threaded programs, it may become a scalability bottleneck.

3. Our Approach

Our approach eliminates heap overflows by building on an alternative to freelist-based allocation. We first describe the heap organization we employ (Section 3.1), present the implementation of a function that efficiently computes remaining space in any heap object (Section 3.2), and show how to use this function to prevent library-based overflows (Section 3.3).

3.1 Heap Organization

We use a heap organization known as *segregated-fits BiBOP-style*. Figure 2 provides a pictorial representation of part of such a heap. The allocator divides memory into chunks that are a multiple of the system page size (typically 4K). This organization into pages gives rise to the name "Big Bag of Pages", or "BiBOP" [15]. BiBOP allocators were originally used to provide cheap access to type data for LISP, but they are also suitable for general-purpose allocation.

In addition to dividing the heap into pages, we require that all objects in the same chunk have the same size — in other words, objects of different sizes are *segregated* from each other. We store the object size and other metadata either at the start of each chunk, or, as shown here, in a page directory table off to the side. By aligning chunks to page boundaries, the allocator can locate the metadata for individual pages in constant time. Masking off the low-order bits of a pointer to the start *or inside* the object yields an index into the page directory.

The page directory indicates whether the given page is dedicated to small objects ("chunk"), if it is the start ("start") or another part ("follow") of a range of pages. It also contains information indicating whether the given page is unmapped, and points to the free list of objects inside a chunk.

What we describe above corresponds to the structure of the BSD allocator, known as PHKmalloc [19], but it is similar to the heap layout of the Hoard scalable memory allocator and Maged's non-blocking variant [7, 5, 21], as well as the Boehm-Demers-Weiser conservative garbage collector [11]. As we describe in Section 3.2, this BiBOP-style heap organization allows us to efficiently prevent library-based heap overflows. In addition, it has numerous advantages over freelist-based schemes:

- **Lower per-object overhead.** In the Lea allocator, an eight-byte object is accompanied by an eight-byte object header, doubling space consumption. By contrast, in the BiBOP organization, one header suffices for an entire pageful of objects, adding just 1% space overhead to these small objects.
- **Eliminates external fragmentation.** Freelist allocators typically allow a region of memory to be allocated to objects of different sizes. Unforuntately, a small allocated object in the middle of a range of free memory *fragments* this space. Segregating object sizes ensures that free space is never broken into smaller pieces, thus preventing *external* fragmentation. *Internal* fragmentation — unused space inside allocated objects — remains a possibility, although it is possible to bound this by choosing appropriate size classes.
- **Reduces virtual memory impact.** In a virtual memory management environment, a BiBOP-style allocator can reduce its working set size. Because it knows when an entire pageful of objects is free, it can invoke the `madvise` call to allow the virtual memory manager to reclaim that page (an equivalent exists on Windows operating systems). This ability reduces an application's physical memory requirements and thus can dramatically reduce paging costs [13, 19].

3.2 Computing Available Space

Figure 3 presents the pseudo-code for the key function we require. This function, called `remainingSize`, takes a pointer as an argument, and returns the remaining space — that is, the number of bytes until the end of the allocated area. We describe below how to implement this operation efficiently in the context of a BiBOP-style allocator; Section 3.3 next shows how this information can be used to prevent library-based heap overflows.

The `remainingSize` function first checks its argument for validity. If it is not on the heap, it returns -1. If the object is already freed or is in an unmapped area, it returns 0. Returning zero prevents inadvertent segmentation violations that would be caused by writing into an unmapped page.

The next step is to find the page information structure corresponding to this pointer (maintained in the array `page_dir`). This page directory points either to a large object that consists of a number of pages, or to a single page. If the object is large, the function iterates through the directory until it finds the object's last page. The size is then just the number of pages traversed minus the offset of the pointer in the page. If the pointer is inside a small object, it just returns the object size minus the offset within the object.

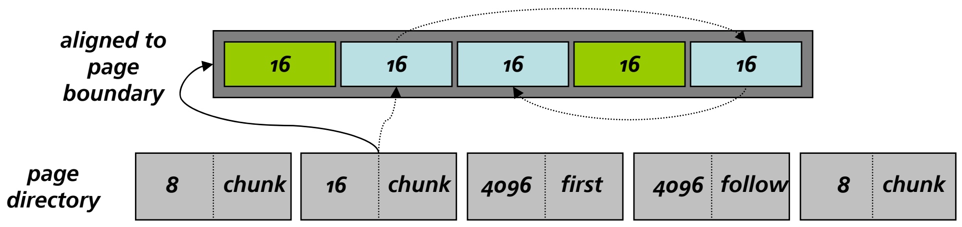


Figure 2. A fragment of a segregated-fits BiBOP-style heap, the basis of the approach presented here. Memory is allocated from page-aligned chunks, and metadata (size, type of chunk) is maintained in a page directory. The dotted lines indicate the list of free objects inside the chunk.

```
size_t remainingSize (void * ptr) {
  // Find index into metadata table.
  index = ptr2index(ptr);
  if (index not on heap)
    return -1;
  if (index to free space)
    return 0;
  // Find page info.
  info = page_dir[index];
  // Compute size remaining.
  if (range of pages) {
    i = total number of pages;
    space = i * pagesize;
    return space - object offset;
  } else {
    space = info->size;
    return space - object offset;
  }
}
```

Figure 3. Pseudo-code for computing remaining size.

Note that this function imposes no space overhead and is efficient. The implementation described here requires no additional bookkeeping because it uses only the existing data structures inside the allocator. For small objects (under 2K), it operates in constant time. For larger objects, it runs in time proportional to the number of pages remaining in the object. For example, computing the size of a 128K object given a pointer to its start requires just 32 directory lookups.

The implementation described here is in the context of PHKmalloc. However, we believe it should be similarly straightforward to apply to similar allocators such as Hoard and the BDW conservative garbage collector.

3.3 Preventing Library-Based Overflows

We are now in a position to alter the definitions of the C library functions. Our library operates by interposition (via `LD_PRELOAD`), and replaces all unsafe functions with checked variants.

Most functions operate similarly to the implementation of `strcpy`, presented in Figure 4. Each function checks to make sure it has a pointer to the actual library implementations, and then computes the remaining size available in any destination pointer. If the size is -1, then the target is on the stack, and any existing stack protection technique could be employed here (e.g., libsafe). otherwise, the remaining size is used either as the total or, in the case of `strncpy`, the maximum amount to be copied.

4. Experimental Results

In this section, we compare the runtime performance of HeapShield to that of the default Linux allocator (based on version 2.7 of the Lea allocator) and to libsafePlus [2] 1.6.0, a system that also prevents library-based heap overflows.

```
char * strcpy (char *dest, const char *src) {
  if (not initialized)
    initialize function pointers;
  sz = remainingSize (dest);
  if (sz == -1) { // not on heap
    // could check for stack overflow here.
    return original_strcpy (dest, src);
  } else {
    s = strlen(src) + 1;
    n = (s < sz) ? s : sz;
    return original_strncpy (dest, src, n);
  }
}
```

Figure 4. Pseudo-code for `strcpy`.

Our experimental platform is a Linux system running version 2.4.20 of the kernel. It is equipped with two Intel Xeon processors, both running at 3.06GHz with hyperthreading enabled. Each processor is equipped with a 512K L2 cache, and the system has three gigabytes of RAM. All programs are compiled at the highest optimization level (-O3) with g++ version 4.0.2. All experiments are performed on a quiescent system, and results are the means of three runs (after one warm-up).

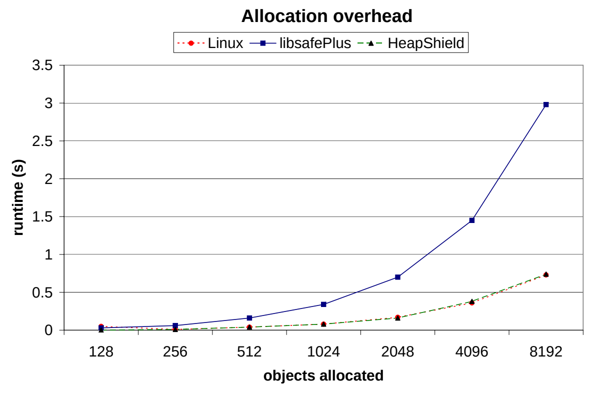
We run two sets of experiments. The first (Section 4.1) is a microbenchmark that isolates the overhead of our approach compared with the runtime performance of both libsafePlus and the Linux allocator, which provides no protection from heap overflows. The second suite (Section 4.2) uses benchmark applications that exercise the memory management system intensively. These benchmark applications have been used in the memory management community to measure the performance of memory managers [8, 12, 14, 18]. While these benchmarks are not necessarily typical of application behavior, they emphasize the overhead of any approach that adds cost to memory operations.

Table 1 describes these benchmarks in more detail, along with their inputs. We use the largest available benchmark inputs for each program, including one of the SPEC2K reference inputs for `perl`.

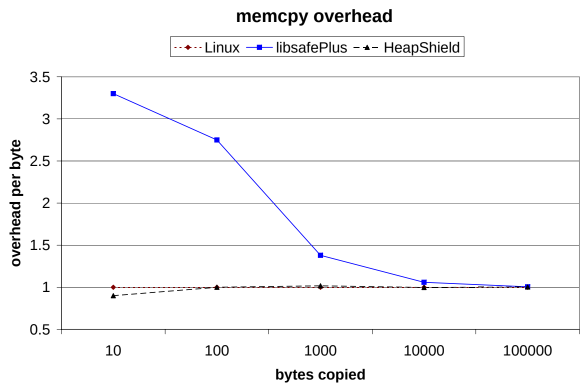
4.1 Microbenchmarks

To compare the bookkeeping overhead of our approach to libsafePlus, we conduct an experiment similar to that presented by Avijit et al˙ [2]. We wrote a synthetic benchmark that repeatedly (1024 times) allocates a given number of eight-byte objects, and then deallocates them all.

Figure 5(a) presents the results. The cost of allocating objects grows linearly with both HeapShield and the Linux allocator. However, because libsafePlus maintains a red-black tree to track object sizes, its cost grows quadratically, raising its complexity to



(a) Allocation overhead (note the x-axis is log-scale). HeapShield matches the performance of the Linux allocator, while libsafePlus's cost grows quadratically.



(b) `memcpy` overhead (note the x-axis is log-scale). libsafePlus is up to 3.3x more expensive than directly calling `memcpy`, although this overhead decreases as object sizes increase. HeapShield matches the Linux `memcpy` performance, regardless of object size.

Figure 5. Performance overhead for microbenchmarks (allocation cost and `memcpy`).

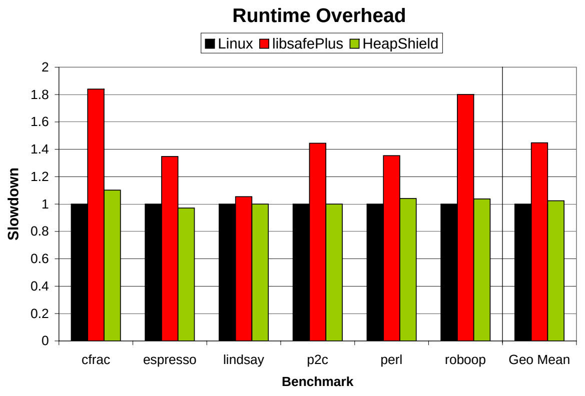


Figure 6. Performance comparison against the Linux allocator, libsafePlus, and HeapShield. While libsafePlus degrades performance by 45% on average, HeapShield's average overhead is just 2.4%.

$O(n^2 \log n)$. Our approach avoids heap overflows without altering the asymptotic complexity of the instrumented program.

We next run a microbenchmark to measure the cost of avoiding heap overflow in library calls. This benchmark performs 1,000,000 `memcpy` operations of a given size. Figure 5(b) presents the overhead per byte, normalized to the Lea allocator. The overhead of using libsafePlus is substantial for small object copying, costing as much as 3.3X per byte to copy 10 bytes, and 2.75X to copy 100 bytes. Our approach and the Linux allocator achieve similar performance regardless of the number of bytes copied.

4.2 Memory-Intensive Benchmarks

We next measure runtime performance across the memory-intensive benchmarks described in Table 1. Figure 6 presents these results, normalized to the Linux allocator.

In general, HeapShield nearly matches the performance of the Linux allocator, which provides no protection from heap overflows. Our system runs no more than 10% slower (for `cfrac`), and as much as 3% faster (for `espresso`); the geometric mean is a slowdown of 2.4%. Notice that this difference in running time is almost entirely due to differences in the underlying memory allocation mechanism, rather than library checks.

However, libsafePlus imposes significant performance overhead for several of the benchmarks. On average, libsafePlus degrades performance by 45% relative to the Linux allocator. This degradation is especially high for three of the benchmarks: 84% for `cfrac`, 44% for `p2c` and 80% for `roboop`.

5. Related Work

There is extensive work on preventing or thwarting buffer overflows. We discuss related work in static analysis techniques used in compiler and language approaches (Section 5.1), and dynamic techniques (Section 5.2), which are most similar to the work presented here.

5.1 Static Approaches

Static analysis techniques can catch a broad range of errors, eliminating the possibility of overflows altogether [29, 3, 30, 1, 22]. Compiler-based approaches are effective but require application source code, are usually restricted to a subset of C (and not C++), and lead to a substantial increase in execution time. The current state of the art is the CRED compiler by Ruwase and Lam [27], which accomodates most of the C language. Full buffer checking yields slowdowns of up to 12X, although checking strings reduces the maximum slowdown to slightly more than 2X.

Rinard et al. present a system built on Ruwase and Lam's CRED compiler that, like our approach, silently drops heap overflows [25]. It is more general than HeapShield, preventing all illegal writes, but suffers from the same performance problems as CRED.

An alternative approach does not avoid heap overflows but uses randomization or obfuscation to make them difficult to exploit [10, 9]. Bhatkar et al. report from 1.12 to 3.37x slowdown in program execution, along with an increase in heap usage ranging from 1x to 5.51x.

5.2 Dynamic Techniques

Several tools, such as Purify [16] and Valgrind [23, 28] can find all buffer overflows at runtime, but their overhead (2-25X) makes them suitable primarily for debugging. Newsome and Song present a dynamic taint analysis tool that detects overwrites using a binary rewriting technique [24]; however, it operates on top of Valgrind and so incurs similar overhead.

It is possible to detect heap overflows after the fact, and then trigger error-handling code. This approach was first proposed by

Benchmark	Description	Input
`cfrac`	Factors arbitrary-length numbers	a 36-digit number
`espresso`	optimizer for programmable logic arrays	`largest.espresso`
`lindsay`	Hypercube simulator	`script.mine`
`roboop`	Robotics simulator	*supplied*
`p2c`	Pascal-to-C translator	`mf.p`
`perl`	Perl interpreter	`perfect.in`

Table 1. Memory-intensive benchmark application suite.

Robertson et al. [26], and has subsequently been implemented in the newest version of the Lea allocator (version 2.8.X).

The most related work to that presented here is libsafePlus [2]. Unlike our approach, which silently truncates overflows and so allows execution to continue, libsafePlus aborts execution when it detects a library-based heap overflow. It uses a red-black tree to track objects allocated from `malloc`. As Sections 4.1 and 4.2 show, this approach adds considerable overhead to `malloc` and `free`, increasing its asymptotic complexity and degrading application performance by up to 84%.

libsafe intercepts library calls to prevent stack-smashing attacks [4]. This approach is orthogonal and complementary to the work presented here.

6. Conclusion

This paper presents HeapShield, an approach that avoids all library-based heap overflows with minimal overhead. It uses an alternative heap layout, known as *segregated-fits BiBOP-style*, that speeds the calculation of object sizes. It then provides alternative definitions of unsafe C library functions that silently truncate any heap overflows. Our approach operates on unaltered programs, requiring no source code or compiler support. It also nearly matches the efficiency of uninstrumented applications, running on average just 2.4% slower (ranging from 3% faster to 10% slower).

We believe that the practicality of HeapShield's approach means it can be immediately and broadly adopted, thus marking the end of the road for library-based heap overflows. In addition to preventing security vulnerabilities, it can eliminate a notorious source of programming errors. It can be used directly, or straightforwardly adapted to extend similar protection to high-performance memory allocators like Hoard and to the Boehm-Demers-Weiser conservative garbage collector. Finally, because its heap protection is orthogonal to approaches that provide stack-based protection, our approach can be combined with these to significantly enhance the reliability and security of deployed applications.

References

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 290–301, New York, NY, USA, 1994. ACM Press.

[2] K. Avijit, P. Gupta, and D. Gupta. Tied, libsafeplus: Tools for runtime buffer overflow protection. In *Proceedings of the 13th USENIX Security Symposium*. USENIX, Aug. 2004.

[3] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 332–341, New York, NY, USA, 2005. ACM Press.

[4] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2000.

[5] E. D. Berger. The Hoard memory allocator. Available at http://www.hoard.org.

[6] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, Cambridge, MA, Nov. 2000.

[7] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, November 2000.

[8] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.

[9] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120. USENIX, Aug. 2003.

[10] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, pages 271–286. USENIX, Aug. 2005.

[11] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[12] D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. *Software Practice and Experience*, 24(6), 1994.

[13] Y. Feng and E. D. Berger. A locality-improving dynamic memory allocator. In *Proceedings of the ACM SIGPLAN 2005 Workshop on Memory System Performance (MSP)*, Chicago, IL, June 2005.

[14] D. Grunwald, B. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, pages 177–186, Albuquerque, NM, June 1993. ACM Press.

[15] D. R. Hanson. A portable storage management system for the Icon programming language. *Software Practice and Experience*, 10(6):489–500, 1980.

[16] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter 1992 USENIX Conference*, pages 125–138, San Francisco, California, 1991.

[17] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, New York, NY, USA, 2004. ACM Press.

[18] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In R. Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 26–36, Vancouver, Oct. 1998. ACM Press.

[19] P.-H. Kamp. Malloc(3) revisited. http://phk.freebsd.dk/pubs/malloc.pdf.

[20] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html,

1997.

[21] M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of SIGPLAN 2004 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Washington, DC, June 2004. ACM Press.

[22] G. C. Necula, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, New York, NY, USA, 2002. ACM Press.

[23] N. Nethercote and J. Fitzhardinge. Bounds-checking entire programs without recompiling. In *SPACE 2004*, Venice, Italy, Jan. 2004.

[24] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 2005 Network and Distributed Systems Security Symposium*, Feb. 2005.

[25] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 2004 Annual Computer Security Applications Conference*, Dec. 2004.

[26] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *LISA '03: Proceedings of the 17th Large Installation Systems Administration Conference*, pages 51–60. USENIX, 2003.

[27] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, page 159.

[28] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, Anaheim, California, USA, Apr. 2005.

[29] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 117–126, New York, NY, USA, 2004. ACM Press.

[30] S. H. Yong and S. Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *ESEC/FSE-11: 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 307–316, New York, NY, USA, 2003. ACM Press.