Archipelago: Trading Address Space for Reliability and Security

Vitaliy B. Lvin Gene Novark
Emery D. Berger
Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
vlvin@cs.umass.edu, gnovark@cs.umass.edu, emery@cs.umass.edu

Benjamin G. Zorn
Microsoft Research
One Microsoft Way
Redmond, WA 98052
zorn@microsoft.com

Abstract

Memory errors are a notorious source of security vulnerabilities that can lead to service interruptions, information leakage and unauthorized access. Because such errors are also difficult to debug, the absence of timely patches can leave users vulnerable to attack for long periods of time. A variety of approaches have been introduced to combat these errors, but these often incur large runtime overheads and generally abort on errors, threatening availability.

This paper presents Archipelago, a runtime system that takes advantage of available address space to substantially reduce the likelihood that a memory error will affect program execution. Archipelago randomly allocates heap objects far apart in virtual address space, effectively isolating each object from buffer overflows. Archipelago also protects against dangling pointer errors by preserving the contents of freed objects after they are freed. Archipelago thus trades *virtual* address space—a plentiful resource on 64-bit systems—for significantly improved program reliability and security, while limiting *physical* memory consumption by tracking the working set of an application and compacting cold objects. We show that Archipelago allows applications to continue to run correctly in the face of thousands of memory errors. Across a suite of server applications, Archipelago's performance overhead is 6% on average (between -7% and 22%), making it especially suitable to protect servers that have known security vulnerabilities due to heap memory errors.

Categories and Subject Descriptors D.2.0 [*Software Engineering*]: Protection mechanisms; D.2.5 [*Software Engineering*]: Error handling and recovery; D.3.3 [*Programming Languages*]: Dynamic storage management; G.3 [*Probability and Statistics*]: Probabilistic algorithms

General Terms Algorithms, Languages, Reliability, Security

Keywords Archipelago, buffer overflow, dynamic memory allocation, memory errors, probabilistic memory safety, randomized algorithms, virtual memory

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.
Copyright © 2008 ACM 978-1-59593-958-6/08/03…$5.00

1. Introduction

Memory errors in C and C++ programs continue to be a significant problem. They are difficult to debug and often easy to exploit. Memory-based attacks are an effective way to compromise Internet servers, either by crashing them, which causes service interruptions and data loss, or by making them execute arbitrary code. Because these bugs are difficult to debug, it can take weeks before even critical errors are repaired [33], leaving applications vulnerable to attack.

A variety of approaches have been developed to help programmers avoid memory errors. These approaches can be roughly classified into three categories: testing tools, garbage collectors, and compiler-based tools. Testing tools, such as Valgrind [24, 31] and Purify [16], impose performance overheads that make their use acceptable only for testing. Conservative garbage collectors [7] protect against dangling pointer errors but provide no protection against buffer overflows. Compiler-based approaches [2, 3, 11, 13, 19, 23, 26, 35, 37] typically incur unacceptably-large runtime overheads or require programmer intervention, and also require source code, which may not be available. They also generally abort program execution in response to memory errors, reducing availability and leaving systems vulnerable to denial-of-service attacks.

Contributions: This paper presents Archipelago, a runtime system that significantly improves the resilience of applications to heap-based memory errors.[1] Archipelago treats heap objects as individual islands, surrounded by stretches of unused address space. On modern architectures, especially 64-bit systems, virtual address space is a plentiful resource. Archipelago trades this plentiful resource for a high degree of *probabilistic memory safety* [4]; that is, Archipelago can use available *virtual* memory to significantly increase the likelihood that a program will run correctly in the face of memory errors.

To control *physical* memory consumption, Archipelago leverages the following key insight: once the distance between objects crosses a certain threshold, each page holds exactly one (small) object. At this point, additional address-space expansion is free: the virtual memory system does not need to allocate physical frames for unused address space between objects. Archipelago takes advantage of this insight and directly allocates one object per page, leaving the virtual address space between objects uncommitted. It

[1] An *archipelago* is an expanse of water with many scattered islands, such as the Aegean Sea.

further limits physical memory consumption by selectively compacting pages of the heap that are infrequently used.

The class of applications that are most sensitive to memory errors and associated security vulnerabilities are servers: they are attractive, high-value targets that are connected directly to the Internet. We show that Archipelago can provide high levels of safety and reliability for this class of applications. We show that Archipelago can let applications run even in the face of thousands of memory errors, while keeping performance impact to acceptably-low levels. Archipelago slows down execution of a range of server applications by just 6% on average (from -7% to 22%). This modest performance impact makes Archipelago a realistic approach to protect deployed server applications against known and unknown heap-based security vulnerabilities.

The rest of the paper is organized as follows. Section 2 reviews operating system support for virtual memory, and explains probabilistic memory safety. Section 3 describes the software architecture of Archipelago in detail. Section 4 evaluates the effectiveness of Archipelago at withstanding memory errors and measures its overhead. Section 5 surveys related work, Section 6 discusses future directions, and Section 7 concludes.

2. Background

2.1 Virtual Memory

Because Archipelago makes extensive use of operating system support for virtual memory management that may not be familiar, we define some important terms and concepts here.

A key distinction is the difference between virtual and physical memory. Virtual memory refers to the full addressable range of memory. Operating systems map virtual memory to available physical memory. On 64-bit systems, virtual memory is plentiful (e.g., 2^{48} bytes on x86-64) while physical memory is in relatively short supply (e.g., on the order of 1–8 gigabytes (2^{30}–2^{33}) bytes).

Virtual memory is divided into pages that are typically 4K chunks. Pages can be in three states: *unmapped*, *reserved*, and *committed*. An unmapped page is not available for use by the process, and access to it triggers a segmentation violation.

When a process obtains a page from the system (via `mmap` in Unix, or `VirtualAlloc` in Windows), the virtual address range is *reserved* so that a subsequent call is guaranteed to return virtual memory from a different range. However, a reserved page does not initially have an associated physical page frame.

When a reserved page is touched for the first time, the page is *committed*: a physical page frame is allocated and associated with the virtual page. The kernel initializes all page contents to zero when they are first touched. Subsequent touches do not result in any page faults unless, due to memory pressure, the page is *evicted* to disk. In this case, the page's contents are generally written to the disk, and then the page is decommitted (but remains reserved). A subsequent touch triggers a page fault, and the kernel will fill the page with the contents previously saved on disk.

Many operating systems allow programmers to direct the kernel's treatment of pages. In Unix, an application can invoke `madvise(MADV_FREE)` to inform the kernel that the data on a range of pages is no longer needed, and thus there is no need to write the contents to disk. In contrast to the `munmap` system call, `madvise(MADV_FREE)` does not unmap the virtual page. If a page is accessed after its contents are discarded, the kernel allocates a fresh, zero-filled page. This call reclaims a page's physical frame, making it available for reuse by the system. Archipelago makes use of `madvise` to limit its physical memory footprint, as Section 3.1 describes. `madvise` can also be used to provide hints to guide the virtual memory manager's page replacement algorithm, a feature that Archipelago also uses.

Additionally, an application can protect access to a page so that accesses trigger a signal, even if the page has been committed. For example, an application can invoke `mprotect(...,PROT_NONE)` on a range of pages: future attempts to read, write, or execute memory on any of these pages will raise a signal. By installing a custom handler to handle these signals, an application can intercept reads or writes to particular pages. Appel and Li describe numerous ways that user-level programs can take advantage of these virtual memory operations [1]. Archipelago uses these calls to manage its compaction and uncompaction of cold objects (see Section 3.2).

2.2 Probabilistic Memory Safety

The motivation for our work comes from the ideas of *infinite heaps* and *probabilistic memory safety* originally introduced by Berger and Zorn and implemented in their DieHard system [4].

An *infinite heap memory manager* is an ideal, unrealizable runtime system that allows programs containing memory errors to execute soundly and to completion. In such a system, the heap area is infinitely large and can never be exhausted. All objects are allocated fresh, infinitely far away from each other, and are never deallocated.

Because every object is infinitely far away from any other object, buffer overflows become benign, and dangling pointers also vanish since objects are never deallocated or reused. A portable correct C program cannot distinguish between an infinite heap memory manager and a normal allocator, while a program containing memory errors would execute correctly for reasons outlined above, as long as it does not contain uninitialized reads.

Of course, it is impossible to build a true infinite heap memory manager. However, one can approximate its behavior by using an M*-heap*—a heap that is M times larger than needed. By placing objects uniformly randomly across an M-heap, we get an expected separation between any two objects of $M-1$ objects, so that smaller overflows become benign, with high probability. By *randomizing* the choice of freed objects to reuse, we minimize the likelihood of recently freed objects being reallocated and subsequently overwritten, and therefore of a malignant dangling pointer error. This heap thus provides *probabilistic memory safety*, a probabilistic guarantee that memory errors occurring in the program are benign during its execution.

In an M-heap, the likelihood of no live objects being overwritten by an overflow N objects in size is $(1-\frac{1}{M})^N$ [4].

This formula shows that one way to increase the probability of correct execution in the presence of memory error is to make the *heap expansion factor* (M) large. For example, $M = 100$ yields a 99% probability that a buffer overflow smaller or equal to the size of an object will be benign. However, DieHard is impractical with large values of M because of its correspondingly large physical memory consumption (see Section 4).

Archipelago achieves these probabilistic guarantees against buffer overflows while consuming only a correspondingly large amount of *virtual* memory. It effectively controls physical memory consumption and provides lower CPU overheads than a comparably-sized DieHard heap, as Sections 4.2 and 4.5 show.

3. Archipelago Architecture

Archipelago consists of three parts: a **randomizing object-per-page memory allocator**, a **hot object space**, and a **cold storage module**, which controls the overall physical memory consumption of the program. Figure 1 illustrates the architecture. These parts are compiled into a dynamically-linked library that, when pre-loaded before an executable, replaces standard memory management routines, such as `malloc` and `free`, with calls to the Archipelago allocator.

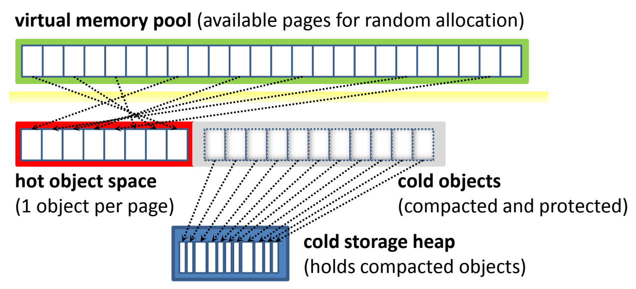

virtual memory pool (available pages for random allocation)
hot object space
(1 object per page)
cold objects
(compacted and protected)
cold storage heap
(holds compacted objects)

Figure 1. Archipelago's software architecture. Archipelago randomly allocates heap objects in virtual address space (Section 3.1). It tracks the hot objects, which are stored one per page (Section 3.2). Cold objects are compacted and placed in cold storage, and the physical memory associated with their page frames is relinquished (Section 3.3).

```
1  void * malloc (size_t size) {
2          void * page = NULL;
3      if (size <= PAGE_SIZE) {
4          //object fits on a page
5          //obtain random page from the pool
6          page = getRandomPage();
7      }
8      if (page == NULL) {
9          //object doesn't fit on the page
10         //or pool is full
11         //mmap memory directly
12         page =
13             mmap(roundUpToPageSize(size),
14                  MAP_ANONYMOUS);
15     }
16     if (page == NULL) {
17         //mmap failed
18         return NULL;
19     }
20     //add coloring
21     void *ptr =
22             getRandomColoring(page, size);
23     //register page(s) as part
24     //of working set
25     registerActivePages(page, ptr, size);
26     return ptr;
27 }
```

Figure 2. Pseudo-code for Archipelago's `malloc`.

```
1  void free (void * ptr) {
2      //retrieve size
3      size_t size = getObjectSize(ptr);
4      //get first page
5      void *page = getStartPage(ptr);
6      //unregister pages being deleted
7      unregisterActivePages(page, ptr, size);
8      //discard pages
9      //that have been compacted
10     discardCompactedPages(page, ptr, size);
11     if (size <= PAGE_SIZE) {
12         //object fits on page:
13         //discard contents
14         madvise(page, MADV_FREE);
15     } else {
16         //object doesn't fit on page:
17         //unmap it
18         munmap(page,
19             roundUpToPageSize(size));
20     }
21 }
```

Figure 3. Pseudo-code for Archipelago's `free`.

3.1 Randomizing Object-Per-Page Allocator

Key to Archipelago's protection from memory errors is its object-per-page memory allocator. It is constructed using the Heap Layers infrastructure [6]. As implied by its name, the object-per-page allocator places each allocated object on a separate virtual memory page. It reserves (but does not commit) a portion of the address space using `mmap`, and uses this space as a pool from which to draw pages to satisfy allocation requests. Figures 2 and 3 present pseudo-code for `malloc` and `free`.

The size of the pool of available pages is a parameter to Archipelago (defaulting to 512 megabytes) that represents the trade-off between the protection Archipelago provides and its virtual memory consumption. A larger pool provides more robust protection against errors, but at the cost of increased virtual memory consumption. Note that under memory pressure, the operating system's virtual memory manager first reclaims any committed but unused pages in the pool, reducing the footprint of the application. Archipelago's physical memory consumption is thus independent of the size of this pool.

Allocation: Objects are placed on pages randomly chosen from the pool (Figure 2, line 6). The object-per-page allocator uses a bitmap to distinguish between used and unused pages. To satisfy allocation requests, it probes the bitmap randomly (`getRandomPage()`) until it finds an unused page. The object-per-page allocator bounds the expected number of probes to find an empty page by keeping the pool no more than half full. This policy bounds the worst-case expected number of probes to a small constant (2).

Because pages in the pool are allocated randomly, no locality of reference exists between different pages. Archipelago uses `madvise`(MADV_RANDOM) to inform the virtual memory manager that no locality exists and that it should not prefetch pages within the pool. Archipelago thus ensures that pages are not instantiated in physical memory until they are actually needed.

To reduce cache conflicts, Archipelago uses *coloring* to place objects on pages. Objects are allocated at random offsets on pages, taking care to keep objects within their pages' boundaries (lines 21–22). Coloring helps reduce L2 misses due to cache conflicts, which can improve performance (see Section 4.4).

Deallocation: When an object smaller than a page in size is deleted, the object-per-page allocator marks the page as free (Figure 3, lines 5–10). Moreover, it instructs the virtual memory manager using `madvise`(MADV_FREE) to discard the contents of the page without writing them to disk, therefore reducing the overhead of the system due to page eviction (line 14).

Large objects: Objects that do not fit on a single page are treated specially by the object-per-page allocator. Archipelago currently does not search for ranges of free pages in the pool but instead allocates memory directly using `mmap` (Figure 2, lines 7–13). When the memory pool becomes more than half full, all objects are allocated via `mmap` to avoid large numbers of repeated probes for free pages in the pool.

Because current Linux kernels randomize locations of memory-mapped objects in the address space, the object-per-page allocator need not take further action. When an object that was allocated using `mmap` is freed, its memory is immediately released back to the operating system using `munmap` (Figure 3, lines 18–19).

3.2 Hot Object Space Management

Running programs with the object-per-page allocator alone would consume so much physical memory that it would be impractical for deployed programs. To limit its physical memory consumption, Archipelago relies on the observed temporal locality of memory accesses in most programs, known as the *working set hypothesis*. At any given time, a program has a *working set*, a subset of the live objects on which the program is actively operating.

The notion of a working set is extensively used in virtual memory managers [10], which attempt to keep the working sets of running programs in memory while evicting rarely used data to disk.

Archipelago follows a similar design. First, the programmer specifies the desired maximum working set size of the program through an environment variable. Archipelago then compacts cold objects not in the working set. It then informs the OS that these now-redundant page frames can be discarded without the need to write them back to disk.

Archipelago keeps all the pages occupied by live objects (the working set) in a bounded FIFO queue. In our current implementation, the size of this FIFO queue is fixed at startup time, either read in from an environment variable or defaulting to 5000 objects. Pages are added to the back of the queue at allocation time. As the queue becomes full, pages at the front of the queue are removed and compacted. Upon access to a compacted page, the page is restored and added to the end of the queue as well.

Our design decision to use FIFO to track the working set is somewhat unconventional. In operating systems, because paging is so expensive, virtual memory managers typically use LRU or CLOCK-based algorithms to manage the working set [8]. These algorithms rely on hardware-managed dirty and reference bits to track information about which pages are in use. However, these bits are maintained by the kernel and are generally unavailable to the user. It is possible to track every page reference in user-space via memory protection mechanisms, but this strategy would impose prohibitively high overhead. An alternative is to modify the operating system, adding a system call to allow users to query page-level dirty and reference bits.

However, for Archipelago, the use of precise algorithms that need to track detailed page activity is unnecessary. Our insight is that the cost of a mistake for Archipelago—compacting and then uncompacting a page that is in the working set—is much lower than the cost of paging to disk. This fact allows us to use a cheap but imprecise approximation such as FIFO to speed the common case. It is more efficient to use a fast algorithm that occasionally goes wrong (with far lower costs than those paid by an OS when paging to disk), rather than using a more intelligent but more expensive strategy.

```
 1  void deflate (void *page) {
 2      // allocate space in cold store
 3      void *coldStore = coldHeap.malloc(
 4          hotPages[page]->getDataSize());
 5      // copy the data
 6      memcpy(coldStore,
 7          hotPages[page]->getDataStart(),
 8          hotPages[page]->getDataSize());
 9      // set trap on future accesses
10      mprotect(page, PROT_NONE);
11      // mark page as cold
12      coldPages[page] = hotPages[page];
13      hotPages.remove(page);
14      // remember the location of the data
15      coldPages[page].
16          setColdStore(coldStore);
17      // return physical page to OS
18      madvise(page, MADV_FREE);
19  }
20
21  bool inflate (void *page) {
22      // check page is valid
23      if (!coldPages.hasKey(page))
24          return false;
25      // enable access to the page
26      mprotect(page, PROT_READ | PROT_WRITE);
27      // restore data
28      memcpy(coldPages[page].getStart(),
29          coldPages[page].getColdStore(),
30          coldPages[page].getSize());
31      // free the cold space
32      coldHeap.free(
33          coldPages[page].getColdStore());
34      // mark page as hot
35      hotPages[page] = coldPages[page];
36      coldPages.remove(page);
37      return true;
38  }
39
40  void sigsegv_handler(void *addr) {
41      if (!inflate(getPageStart(addr))) {
42          // Access outside heap
43          abort();
44      }
45  }
```

Figure 4. Pseudo-code for Archipelago's compaction and uncompaction routines (Section 3.3).

3.3 Cold Storage

Archipelago compacts pages not in the current working set, thus reducing its physical memory requirements. It uses an in-memory compaction mechanism that stores compacted objects in a separate heap managed by a standard allocator (the Lea allocator [21]).

When Archipelago compacts a page, it recomputes the actual size of the object on that page by scanning it backwards until the first non-zero word. It then copies all of the non-zero contents—which may contain a buffer overflow—into the internal heap (Figure 4, lines 2–8). It next disables direct access to the page by removing read and write access via `mprotect`(line 10), so that Archipelago receives a signal the next time the application tries to access the page. Finally, Archipelago removes the page from the hot space and calls `madvise` to instruct the virtual memory manager to discard the page contents to disk rather than write them to disk (lines 11–18).

Archipelago installs a custom signal handler that receives segmentation violation signals and manages restoring objects from cold storage on demand (Figure 4, lines 40–45). When the handler receives a signal, it first checks if the access is a true segmentation violation, which terminates the program. However, if the application was trying to access an object in cold storage, the handler "inflates" the object. The handler first unprotects the page and copies the data back from cold storage (Figure 4, lines 26–30). It also places the page back in the hot space, and frees the space used to hold the object in cold storage (Figure 4, lines 32–36). Control then passes back to the application, which can now safely continue.

While compacting pages imposes additional runtime overhead, it effectively controls physical memory overhead, as Section 4.5 shows.

4. Evaluation

In our evaluation, we answer the following questions:

1. What is the runtime overhead of using Archipelago?
2. What is the memory overhead of using Archipelago?
3. What is the effect of changing Archipelago's heap and pool sizes?
4. How effective is Archipelago against both injected faults and real errors?

4.1 Experimental Methodology

We perform our evaluation on a quiescent dual-processor with 8 gigabytes of RAM. Each processor is a 4-core 64-bit Intel Xeon running at 2.33 Ghz and equipped with a 4MB L2 cache.

We compare Archipelago to the GNU C library, which uses a variant of the Lea allocator [21], and to DieHard, version 1.1. This version, available from the project website, is an adaptive variant that dynamically grows its heap [5], and so is more space-efficient than the original, published description [4].

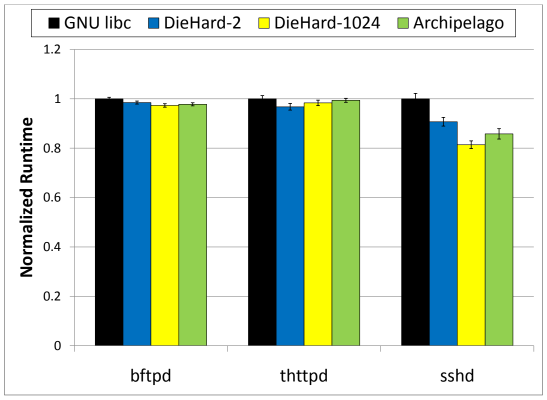
One important caveat is that we run all experiments on a particular version of a recent Linux kernel, version 2.6.22-rc2-mm1. This kernel version uses a more sophisticated algorithm for managing physical memory pages that were initially used by applications, but then returned to the kernel. This *page laundering* process updates a number of kernel data structures and potentially writes the page's contents to secondary storage. Linux kernel versions up to and including 2.6.23 launder pages *eagerly* whenever an application calls `madvise`. However, Linux version 2.6.22-rc2-mm1 launders pages *lazily*, waiting until more physical memory pages are actually needed. Without memory pressure, this policy doubles our system's performance on a memory-intensive microbenchmark, because `madvise` is on Archipelago's normal deallocation path. Because of its performance advantages for ordinary workloads (e.g. MySQL), we expect that this patch, or one similar to it, will be adopted in future versions of the Linux kernel.

4.2 Server Application Performance

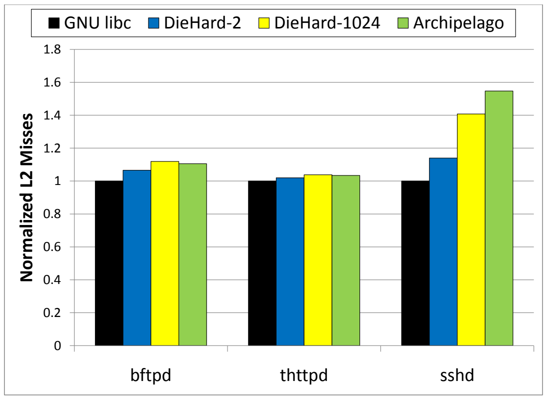
To quantify the performance overhead of using Archipelago, we measure the runtime of a range of server applications running with and without Archipelago. In our experiments, Archipelago uses a memory pool of 512 megabytes, when not otherwise stated. We also compare performance against DieHard with two different heap multiplier values: 2 and 1024. The first multiplier provides performance and protection similar to the results reported in the original DieHard paper, while the second multiplier more closely approximates the level of protection that Archipelago achieves.

We use three different server applications: the *thttpd* web server, the *bftpd* ftp server, and the *OpenSSH* server. For the first two, we record total throughput achieved with 50 simultaneous clients issuing 100 requests each. For OpenSSH, we record the time it takes to perform authentication, spawn a shell, and disconnect. We run each benchmark 10 times and report the mean and its 95% confidence interval.

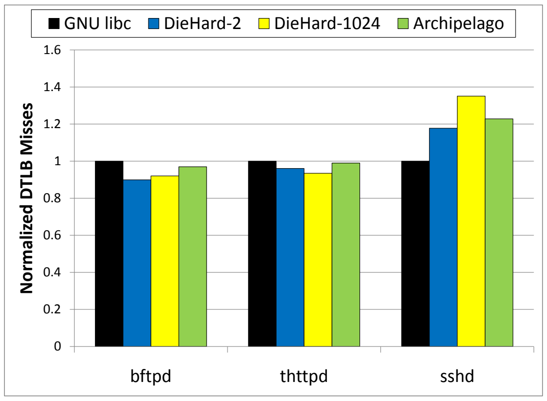
We focus on the CPU impact of our benchmarks by performing all our experiments over the loopback network interface, so that any performance impact is not swamped by network latency. These measured runtime overheads are thus conservative estimates of the performance overhead one would see in practice.



(a) Runtime



(b) L2 Misses



(c) DTLB Misses

Figure 5. Performance across a range of server applications (Section 4.2), normalized to GNU libc (smaller is better).

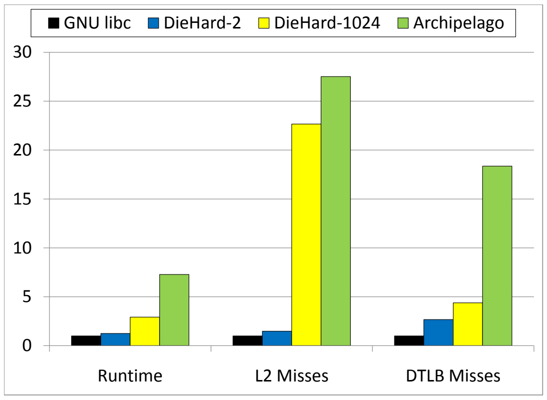


Figure 6. Performance metrics for the memory-intensive *espresso* benchmark (Section 4.3), normalized to GNU libc (smaller is better).

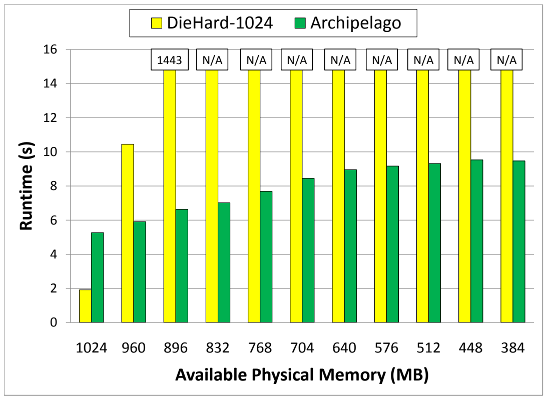


Figure 7. Runtime of the memory-intensive *espresso* benchmark under memory pressure (Section 4.3).

Table 2 presents the allocation characteristics of the servers in our benchmark suite. Because Archipelago's allocator both uses more CPU time and more space than conventional allocators, its time and space overheads are dependent on the number of heap allocations during the program and the number of live objects. These server benchmarks have low allocation rates and few live objects, keeping Archipelago's overhead low.

Figure 5 presents the results of these experiments, normalized to GNU libc. These results show that Archipelago can protect servers without sacrificing server performance. Archipelago's runtime overhead is less than 3% for *bftpd* and *thttpd*, and 17% for OpenSSH. Because these applications never use a large amount of live memory, the number of L2 and TLB misses is low for every allocator. This result shows that neither DieHard nor Archipelago hurt memory system performance for these server applications.

4.3 Memory-Intensive Program Performance

To evaluate the worst-case overhead one could expect for Archipelago, we also measure the performance impact of Archipelago on an extremely memory-intensive benchmark, *espresso*. *Espresso* allocates and deallocates approximately 1.5 million objects in less than a second. This allocation rate greatly exceeds that of a typical server application. In our experiments, we run *espresso* with the same allocators we use in our server experiments.

Figure 6 shows the runtime and number of L2 and DTLB misses of *espresso* with all the memory managers, normalized to GNU libc. As expected, Archipelago's impact on *espresso*'s runtime is significantly higher than on the server applications. Compared to GNU libc, *espresso* runs 1.24, 2.92 and 7.32 times slower with DieHard-2, DieHard-1024 and Archipelago, respectively. However, as Figure 7 shows, Archipelago's ability to control its working set size yields far better performance than DieHard-1024 in the presence of memory pressure. As available memory decreases from 1GB to 384MB, *espresso* running with Archipelago takes between 5.27s and 9.47s. With DieHard-1024, its runtime spikes to 1443 seconds (more than 24 minutes) at 896MB available, and does not run in any reasonable time for smaller amounts of available physical memory.

4.4 Impact of Coloring

As described in Section 3.1, Archipelago's object-per-page allocator uses coloring to reduce cache conflicts. Somewhat surprisingly, the impact of coloring is undetectable on both the server benchmarks and *espresso*. However, it dramatically improves performance on an adversarial microbenchmark. This program allocates 4096 small objects and repeatedly reads them in order of allocation. Without coloring, each access causes a cache miss because all of the objects map to a few sets in the cache. With random coloring, performance improves significantly as the entire cache is utilized, running almost 3 times faster than the version without coloring. Since this optimization offers the potential to substantially improve performance but does not degrade performance for any of the benchmarks, we leave it enabled.

4.5 Space Overhead

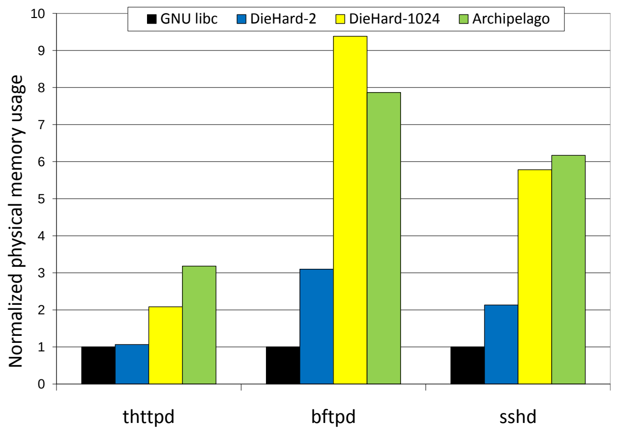
We evaluate the additional memory consumption incurred by using Archipelago, and compare this to DieHard and GNU libc, both with and without memory pressure. We simulate memory pressure by locking an increasing amount of memory until the application's working set no longer fits in physical memory, and report that number as the working set size.

Figures 8(a) shows the resident memory consumption of *thttpd*, *bftpd*, and *sshd* without memory pressure. Note that unlike the other allocators, Archipelago preallocates a large memory pool at startup, increasing its virtual memory consumption. A large fraction of that allocated space—more than 70%—is never actually committed to memory. This effect inflates Archipelago's apparent resident set size, which ranges from 3.18 to 7.87 times as much as with GNU libc, making it comparable to DieHard-1024.

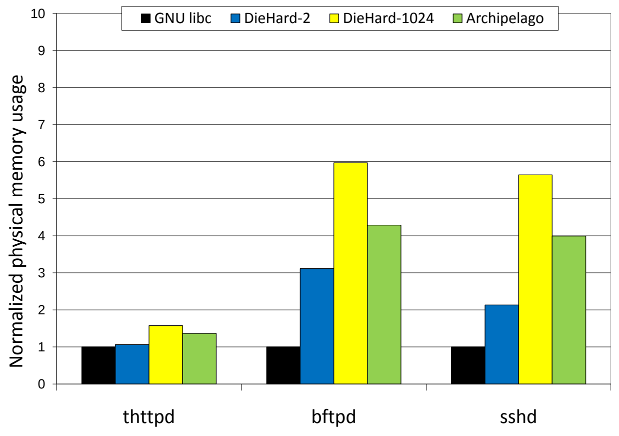
However, Figure 8(b) reveals that under memory pressure, the amount of actual physical memory needed with Archipelago is strictly less than with DieHard-1024. For *thttpd*, *bftpd*, and *sshd*, Archipelago consumes 1.37, 4.29, and 3.99 times as much memory as GNU libc, while DieHard-1024 consumes 1.57, 5.97, and 5.64 times as much.

4.6 Address Space and Hot Space Sizing

Archipelago's performance is dependent on two user-supplied parameters: the size of the virtual address space used for allocation, as well as the size of the hot object space (i.e., the maximum number of uncompacted pages). Increasing the amount of virtual address space available increases the effectiveness of Archipelago's buffer overflow protection, but at the possible cost of degraded TLB performance and increased page table overhead. Increasing the num-



(a) Resident memory usage, without memory pressure.



(b) Resident memory usage, with memory pressure.

Figure 8. Resident memory usage with and without memory pressure (Section 4.5), normalized to GNU libc. Under memory pressure, Linux quickly reclaims Archipelago's uncommitted pages, making its physical memory consumption strictly lower than with DieHard-1024.

ber of pages used for hot objects reduces overhead due to object compaction, but significantly increases memory overhead.

In order to explore these tradeoffs, we performed experiments varying these parameters for *espresso*. Figure 9 shows how these parameters affect execution time. Varying the hot space size has predictable results: too small a space (128 MB) significantly degrades performance because the working set does not fit, leading to repeated compaction and uncompaction of hot objects. Increasing to 256MB captures the working set, so increasing the hot space to 512MB has little effect.

Increasing the amount of virtual address space available to Archipelago shows a consistent trend. A larger virtual address space has little impact on user time, but results in increasing time spent in the kernel. This time is due to a poor fit between the requirements of Archipelago and the current design of Linux's internal data structures, which tend to grow linearly as the number of pages that are randomly protected and unprotected grows.

4.7 Avoiding Injected Faults

We evaluate the effectiveness of Archipelago in tackling memory errors by using two different types of fault injectors: an overflow injector and a dangling pointer injector. We inject faults into *espresso* running with GNU libc, DieHard and Archipelago. We perform all our injection experiments 100 times, and record the number of times that *espresso* produces correct output. Table 1 summarizes these results.

Buffer overflows: We perform three sets of experiments with the overflow injector. We inject 8-byte overflows with 0.01 probability, 4K overflows with 0.001 probability, and 8K overflows with 0.0001 probability. These probabilities correspond to thousands, hundreds, and tens of injected faults, respectively.

In this set of experiments, GNU libc crashes every time, as expected. Archipelago substantially outperforms both variants of DieHard across the range of overflow sizes and frequencies. With small and frequent overflows, Archipelago runs correctly every time. DieHard-1024 does reasonably well, running correctly 77% of the time, while DieHard-2 only runs correctly 29% of the time.

With large but infrequent overflows, Archipelago runs correctly 68% of the time. In this case, DieHard-1024 runs correctly only 23% of the time, while DieHard-2 crashes every time. Even in the worst case of large and reasonably frequent overflows, Archipelago lets `espresso` run correctly 42% of the time, while it only runs 2% of the time with DieHard-1024 (DieHard-2 crashes every time in this case).

These results show that Archipelago provides excellent protection against buffer overflows and offers dramatic improvement over DieHard, even with an expansion factor of 1024.

Dangling pointers: Archipelago's design goal was to limit the impact of buffer overflows, but it also provides a measure of protection against dangling pointers. To measure the impact of dangling pointers on runtime systems, we injected dangling pointer faults that free objects 5, 10 and 20 allocations early with probabilities 0.01, 0.001 and 0.0001, respectively.

These experiments show that, as expected, DieHard-1024 offers better protection from dangling pointer errors than Archipelago: it has vastly more available object slots for reuse. Archipelago has fewer potential slots to place new objects, since it only allows one object per page. Archipelago also instructs the operating system that all freed objects are available for the operating system to reuse at its discretion. If the operating system reuses a page, the original contents will be lost, and access through a dangling pointer to this data will trigger a fault (effectively detecting, but not correcting, the error). Nonetheless, Archipelago provides substantial protection against these errors, running correctly 29% of the time in the first experiment, 67% of the time in the second, and 98% in the third.

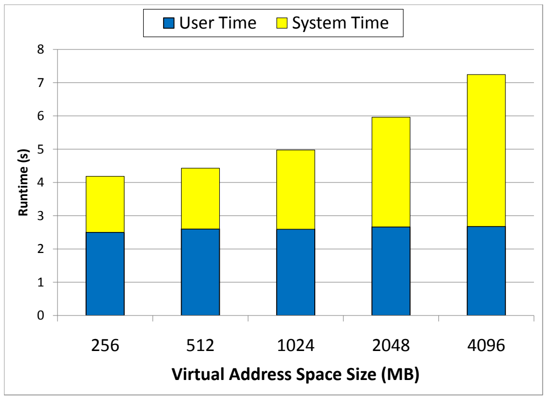
4.8 Avoiding Real Buffer Overflows

To evaluate the effectiveness of Archipelago against real-life buffer overflows, we reproduce two well-known buffer overflow-based exploits: one in the *pine* mail reader, and the other in the *Squid* web cache proxy.

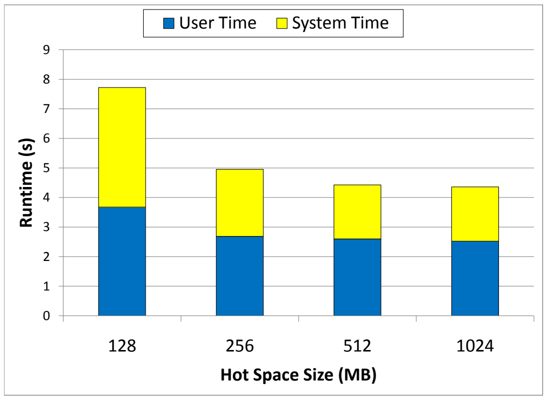
We reproduce an exploit in *pine* version 4.44. The exploit is a buffer overflow that can be triggered by a malformed email message and causes *pine* to crash and fail to restart until the message is manually removed. When we place a malformed message in a user's mailbox, *pine* with GNU libc crashes whenever the user attempts to open that mailbox. However, when running with Archipelago, *pine* successfully opens the mailbox and performs all standard operations with messages in it, including the malicious message, without any user-noticeable slowdown.

Injection experiments (% correct executions)				
espresso	**GNU libc**	**DieHard-2**	**DieHard-1024**	**Archipelago**
		buffer overflows		
8 bytes, $p = 0.01$	0%	29%	77%	100%
8K, $p = 0.0001$	0%	0%	23%	68%
4K, $p = 0.001$	0%	0%	2%	42%
		dangling pointers		
5 mallocs, $p = 0.01$	0%	8%	91%	29%
10 mallocs, $p = 0.001$	0%	75%	100%	67%
20 mallocs, $p = 0.0001$	0%	96%	100%	98%

Table 1. The performance of various runtime systems in response to injected memory errors (Section 4.7). Archipelago provides the best protection against overflows of all sizes and frequencies, and reasonable protection against dangling pointer errors (all executions fail with GNU libc).



(a) Varying virtual address space



(b) Varying hot space size

Figure 9. Impact of sizing parameters on *espresso* runtime (Section 4.6).

We also test Archipelago's ability to withstand a heap buffer overflow for the *squid* web cache. For version 2.3.STABLE5, a maliciously formed request causes a buffer overflow that corrupts heap meta-data (this causes GNU libc to terminate). When running with Archipelago, *squid* consistently handles the malicious request correctly, without crashing.

5. Related Work

This section first discusses past work that exploits large address spaces, and then describes related work in the spheres of memory management, fault tolerance, and software engineering that address the problem of memory errors in C/C++ programs.

The advent of 64-bit processors sparked research in operating systems designed for large address spaces [9]. Druschel and Peterson point out that this address space is sufficiently large that it can be used to provide high performance protection and security by hiding processes from each other [15]. Anonymous RPC (ARPC) uses random placement of processes in a large address space to eliminate expensive hardware context switches on cross-domain RPC calls [36]. Archipelago also leverages a large address space, but instead of using the space to protect independent processes from each other, it isolates individual objects from memory errors within the same process.

Archipelago builds on the ideas of Berger and Zorn's DieHard system [4]. Like DieHard, Archipelago uses a randomized memory manager to provide protection from buffer overflows and dangling pointer errors. Unlike DieHard, Archipelago achieves high reliability by dramatically increasing the size of the address space and does not use replication. By exploiting both standard OS mechanisms and common program behavior, Archipelago provides greater resilience to buffer overflow errors with moderate and acceptable CPU and memory overhead.

Exterminator is another runtime system that, like DieHard, is based on randomized, over-provisioned heaps [25]. The focus of Exterminator is on automatic error detection and correction based on accumulating data from multiple executions. While Archipelago can also be used for overflow detection, it is closer in spirit to DieHard, and unlike Exterminator, provides greater error tolerance without the requirement that errors first be detected.

Numerous compiler-assisted approaches have been introduced to combat memory errors. Semantics provided by Archipelago to programs containing buffer overflows are similar to those of Rinard et al.'s Boundless Memory Blocks [29]. Because Boundless Memory Blocks uses a fixed-size LRU cache to store the values of out-of-bounds writes, accesses to out-of-bounds addresses are undefined if the object has been evicted from the cache. Several other unsound approaches have been proposed [14, 30]. Dhurjati et al. use pool allocation to provide an efficient form of memory safety that guarantees that structure fields are referenced with the correct type. While they guarantee type-safety, there is no guarantee that the object the programmer had intended to access is correctly accessed [14]. Unlike this previous work, Archipelago provides a strong, quantifiable probabilistic guarantee that the intended program behavior will be preserved.

More traditional safe-C compilers [32, 23, 26] use modified versions of C and some combination of static analysis and dynamic checks to provide protection from memory errors. Cyclone [19, 32] augments C with an advanced type system to provide safe explicit memory management. CCured [23] inserts dynamic checks into the compiled program and uses static analysis to eliminate checks from places where memory errors cannot occur. CRED [26] only targets string buffer overflows, and inserts dynamic checks on memory accesses that use out-of-bounds pointers. All of these techniques are aimed at detecting memory errors and terminating the program in response. Archipelago, on the other hand, is aimed at avoiding memory errors and allowing the program to continue running correctly.

Like Archipelago, Rx can help avoid memory errors [28]. It performs periodic checkpointing of program execution, and when an error occurs, it re-runs the program from a checkpoint in a modified environment. In response to crashes, Rx pads allocations to avoid buffer overflows, and delays reuse of freed memory to prevent dangling pointers. Two fundamental limitations of Rx are that it only works with applications that allow replay, and cannot cope with errors that do not result in crashes. Archipelago does not suffer from either of these limitations.

Dangling pointer errors have been addressed in several ways in previous work. Dhurjati et al. employ a clever use of virtual memory page mapping and protection to allow them to detect dangling pointers at low cost [12]. While Archipelago also uses virtual memory protection, our focus is on providing resilience to buffer overflows with less emphasis on dangling pointers. Garbage collection is an alternative runtime system that provides safety from dangling pointer errors. The most commonly used garbage collector for C programs is Boehm-Demers-Weiser conservative garbage collector [7]. Unlike Archipelago, garbage collection provides no protection against buffer overflow errors. Garbage collection also imposes significant space and time overheads to achieve reasonable performance [17].

Finally, various testing tools and debugging allocators [20, 22, 27] can aid programmers in debugging memory errors. They incur prohibitively high overhead both in terms of performance (up to 25X) and space (10X), making them only suitable for testing. Valgrind [24, 31] and Purify [16] use binary instrumentation or emulation to detect memory errors at runtime. Electric Fence [27] is a debugging allocator that, like Archipelago, allocates heap objects on separate pages. It allocates three pages for every object: one page for the object itself (placed at the end), and a memory-protected page before and after the object. Unlike Archipelago, Electric Fence does not perform compression and aborts whenever an overflow causes a memory protection fault.

6. Future Work

There are several ways that the current Archipelago implementation could be improved. One possibility is to not just compact cold objects but to compress them, which offers the potential to further minimize Archipelago's memory overhead. While intuitively appealing, the use of compression adds complexity and requires adaptive algorithms to keep its CPU overhead low [34].

We intend to explore adaptively sizing the memory pool size to achieve the optimal trade-off between performance overhead and resilience to errors. Our current implementation has a static FIFO size, and we intend to investigate techniques to grow and shrink the FIFO size just as an OS virtual memory manager adapts working set size.

Because our approach uses very large virtual address spaces with sparsely mapped pages, we plan to investigate how OS support for sparse page tables could improve Archipelago's performance. In addition, hardware TLBs have remained relatively small, despite the enormous growth in physical memory sizes over the last two decades. We anticipate that TLB designs that better accomodate large sparse virtual memories, such as those proposed by Huck and Hays [18], would significantly benefit Archipelago's performance.

7. Conclusion

Archipelago is a runtime system that provides protection from memory errors for unmodified C programs. It provides probabilistic protection from both buffer overflows and dangling pointer errors with high probability. Archipelago spreads objects far apart in the address space and randomizes the choice of freed objects to reuse, giving applications an illusion of infinite-size heap and protecting them from memory errors. It leverages the virtual memory subsystem of the underlying OS to efficiently provide a high level of memory safety to target programs at low cost.

We show that Archipelago increases the resilience of programs to both real and injected memory errors. Archipelago allows programs to correctly execute through hundreds and even thousands of memory errors, which is a significant improvement over current state-of-the-art systems.

We further demonstrate that the overhead of using Archipelago is more than acceptable across a range of different server applications, both in terms of CPU performance and memory usage. We believe Archipelago is especially suitable for deployment to protect servers that have known security vulnerabilities due to heap memory errors.

8. Acknowledgments

The authors would like to thank all of the anonymous reviewers for their careful and detailed reviews. We would also like to thank Shan Lu and Yuanyuan Zhou for providing us the buggy inputs for *squid*, and Martin Rinard for the *pine* exploit.

References

[1] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 96–107, New York, NY, USA, 1991. ACM Press.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 290–301, New York, NY, USA, 1994. ACM Press.

[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] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006)*, pages 158–168, New York, NY, USA, 2006. ACM Press.

[5] E. D. Berger and B. G. Zorn. Efficient probabilistic memory safety. Technical Report UMCS TR-2007-17, Department of Computer Science, University of Massachusetts Amherst, Mar. 2007.

[6] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001)*, Snowbird, Utah, June 2001.

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

Benchmark	Max live bytes	Max live objects	Total bytes allocated	Total objects allocated	Alloc rate (bytes/s)
bftpd	142,723	713	42,057,815	41,159	6,051,484
thttpd	342,280	705	45,255,581	40,791	3,242,810
sshd	568,304	5,203	6,596,222	30,437	1,945,214

Table 2. Server benchmark characteristics: maximum live size, total allocated memory over the life of the program, and allocation rate.

[8] R. W. Carr and J. L. Hennessy. Wsclock - a simple and effective algorithm for virtual memory management. In *SOSP*, pages 87–95, 1981.

[9] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, Nov. 1994.

[10] P. J. Denning. The working set model for program behaviour. *Communications of the ACM*, 11:323–333, 1968.

[11] D. Dhurjati and V. Adve. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 2006 International Conference on Software Engineering (ICSE'06)*, Shanghai, China, May 2006.

[12] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 269–280, Washington, DC, USA, 2006. IEEE Computer Society.

[13] D. Dhurjati, S. Kowshik, and V. Adve. Safecode: enforcing alias analysis for weakly typed languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 144–157, New York, NY, USA, 2006. ACM Press.

[14] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, San Diego, CA, June 2003. ACM Press.

[15] P. Druschel and L. L. Peterson. High-performance cross-domain data transfer. Technical Report TR 92-11, Dept. Comp. of Sc., U. of Arizona, Tucson, AZ (USA), Mar. 1992.

[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. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, San Diego, CA, Oct. 2005.

[18] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 39–50, New York, NY, USA, 1993. ACM Press.

[19] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[20] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic. Comprehensively and efficiently protecting the heap. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 207–218, New York, NY, USA, 2006. ACM Press.

[21] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1997.

[22] Microsoft Corporation. Pageheap. http://support.microsoft.com/kb/286470.

[23] 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.

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

[25] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2007. ACM Press.

[26] O. Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, Feb. 2004.

[27] B. Perens. Electric Fence v2.1. http://perens.com/FreeSoftware/ElectricFence/.

[28] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies: A safe method to survive software failures. In *Proceedings of the Twentieth Symposium on Operating Systems Principles*, volume XX of *Operating Systems Review*, Brighton, UK, Oct. 2005. ACM.

[29] 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.

[30] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Sixth Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004. USENIX.

[31] 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.

[32] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory management in cyclone. *Science of Computer Programming*, 2006. Special issue on memory management. Expands ISMM conference paper of the same name. To appear.

[33] Symantec. Internet security threat report. http://www.symantec.com/enterprise/threatreport/index.jsp, Sept. 2006.

[34] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the Annual Technical Conference on 1999 USENIX Annual Technical Conference*, pages 101–116, Berkeley, CA, USA, 1999. USENIX Association.

[35] 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.

[36] C. Yarvin, R. Bukowski, and T. Anderson. Anonymous RPC: Low-latency protection in a 64-bit address space. In *Proceedings of the 1993 Summer USENIX Conference*, pages 175–186, 1993.

[37] 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.