2-2011

Hardening Software Against Memory Errors and Attacks

Albert Eugene Novark
University of Massachusetts Amherst

HARDENING SOFTWARE AGAINST MEMORY ERRORS AND ATTACKS

A Dissertation Presented

by

ALBERT EUGENE NOVARK

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2011

Department of Computer Science

HARDENING SOFTWARE AGAINST MEMORY ERRORS AND ATTACKS

A Dissertation Presented

by

ALBERT EUGENE NOVARK

To BK Reeves.

ACKNOWLEDGMENTS

I'd first like to thank Emery Berger, my thesis advisor, for his unwavering enthusiasm and support. His influence has taught me to always approach research pragmatically, but also to explore the ideas that *just can't* work. I also thank my dissertation committee: Scott Kaplan, Israel Koren, and Yannis Smaragdakis, for their valuable insights, feedback, and support.

I am fortunate to have worked with Ben Zorn, Cliff Click, Sam Guyer, Kathryn McKinley, and Erik Learned-Miller, who provided valuable guidance on projects we worked on together, but more importantly, taught me different ways of thinking about research and solving problems. Leeanne Leclerc and Laurie Downey deserve special thanks for making my experience at UMass go as smoothly as possible.

I couldn't have gotten any of this done without the help of fellow PLASMA labmates, including Vitaliy Lvin, Ting Yang, Matthew Hertz, Tongping Liu, Charlie Curtsinger, and Justin Aquadro. I am also grateful for sharing an office with Trek Palmer, Tim Richards, and Ed Walters, who provided both support and much-needed distraction.

I'm indebted all the friends I've made during graduate school. I'd particularly like to thank George Konidaris, Sarah Osentoski, Steve Murtagh, Ricky Chang, Aaron St. John, TJ Brunette, Cheryl Caskey, and Scott Niekum. Everyone else: forgive me for forgetting you; you know I'm writing this at the very last minute.

Finally, I thank my parents, sister, and the rest of my family for their unconditional love and support, even when I got the crazy idea to get a Ph.D.

ABSTRACT

HARDENING SOFTWARE AGAINST MEMORY ERRORS AND ATTACKS

FEBRUARY 2011

ALBERT EUGENE NOVARK

B.S., UNIVERSITY OF TEXAS

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Emery D. Berger

Programs written in C and C++ are susceptible to a number of memory errors, including buffer overflows and dangling pointers. At best, these errors cause crashes or performance degradation. At worst, they enable security vulnerabilities, allowing denial-of-service or remote code execution. Existing runtime systems provide little protection against these errors. They allow minor errors to cause crashes and allow attackers to consistently exploit vulnerabilities.

In this thesis, we introduce a series of runtime systems that protect deployed applications from memory errors. To guide the design of our systems, we analyze how errors interact with memory allocators to allow consistent exploitation of vulnerabilities. Our systems improve software in two ways: first, they tolerate memory errors, allowing programs to continue proper execution. Second, they decrease the probability of successfully exploiting security vulnerabilities caused by memory errors. Our first system, Archipelago, protects exceptionally sensitive server applications against severe errors using an object-per-page randomized allocator. It provides near-100% protection against most buffer overflows. Our second system, DieHarder, combines ideas from Archipelago, DieHard, and other systems to enable maximal protection against attacks while incurring minimal runtime and memory overhead. Our final system, Exterminator, automatically corrects heap-based buffer overflows and dangling pointers without requiring programmer intervention. Exterminator relies on both a low-overhead randomized allocator and statistical inference techniques to automatically isolate and correct errors in deployed applications.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Memory errors, including buffer overflows, dangling pointers, and memory leaks, remain a leading cause of program crashes and security vulnerabilities. These errors affect almost all applications, including internet servers, desktop applications, and embedded systems. Managed programming languages such as C# and Java provide *memory safety*, detecting overflows by performing bounds checks and preventing dangling pointers by using type-safe garbage collection. However, many commonly-used applications are written in unmanaged languages such as C and C++ that provide little protection against these errors.

Ideally, even unmanaged language implementations would handle these errors gracefully. Errors should be *tolerated* whenever possible: overflows should not corrupt useful data, and dangling pointers and memory leaks should be prevented by garbage collection. When toleration is not possible, programs should *resist* security vulnerabilities by eliminating the possibility of successful exploitation of errors.

Techniques such as bounds-checking and garbage collection that deterministically ensure memory safety for Java and related languages have been applied to C/C++. Unfortunately, some aspects of C/C++ complicate the implementation of these techniques. Bounds checking requires validating all pointer arithmetic, slowing down program execution by an order of magnitude. Garbage collection prevents dangling pointer errors and many memory leaks. However, precise garbage collection requires reliable distinction of pointer and non-pointer values, which is impossible in C. One solution is conservative garbage collection, but it cannot reclaim all unused memory and has unpredictable performance. Such systems thus rarely used in C/C++ production systems because of these drawbacks.

While deterministic memory safety may require unacceptable tradeoffs, *probabilistic* approaches sacrifice 100% protection in order to maintain acceptable runtime and memory overheads. Systems implementing probabilistic memory safety provide sound probabilistic guarantees about error tolerance. Existing systems, including those described in this dissertation, rely on randomized runtime systems to achieve these guarantees. Berger and Zorn first introduced the concept of probabilistic memory safety and implemented DieHard [9], which provides probabilistic protection against buffer overflows and dangling pointer errors.

By definition, probabilistic memory safety does not tolerate all errors. When tolerance fails, the runtime system should limit the effect of the error to prevent security vulnerabilities. In the context of security, the failure modes of the runtime system are as important as its ability to fully tolerate errors. Our systems provide probabilistic protection against attacks by decreasing the probability of attack success when error tolerance fails.

In some cases, error tolerance and vulnerability resistance are in tension. The runtime system should prevent program crashes due to benign programming errors while simultaneously preventing exploitation of errors, even if the latter goal requires terminating the program.

Contributions

In this dissertation, we focus on the interactions between memory allocators and memory errors, which often cause crashes and security vulnerabilities. We describe several systems that increase error tolerance and decrease attack effectiveness when compared to existing memory allocators, including DieHard.

Specifically, this thesis makes the following contributions:

- We analyze the interaction between several widely-used allocators and memory errors. We show that conventional allocator designs are inherently vulnerable to a number of attack strategies. We analyze the success probability of several attack strategies against both traditional and randomized allocators and show that predictability is the inherent weakness that enables consistent successful attacks. Furthermore, we show that allocator features that increase reliability can have a negative impact on security.

Building on this analysis, we implement three systems that provide enhanced security and reliability over the state-of-the-art. Each system provides sound probabilistic guarantees about error tolerance and security, which we analyze and prove. Specifically:

- We present a runtime system called **Archipelago** that deterministically tolerates limited overflows while providing strong protection against larger overflows and dangling pointers. Archipelago's design fully protects against contiguous buffer overflow exploits. We analyze Archipelago's error and attack tolerance and show that it provides much greater protection than DieHard.

  To provide this enhanced protection, Archipelago allocates one object per virtual page and relies on virtual memory protection to enable compaction of stale objects. Archipelago has low runtime and memory overhead for applications with low heap footprint, such as small server applications, but is not suitable for general-purpose use.

- We present **DieHarder**, a memory allocator that combines ideas from DieHard, Archipelago, and the OpenBSD allocator to provide maximal protection against attacks while incurring minimal runtime and memory overheads. We show empirically that DieHarder provides its enhanced security with modest performance overhead (around 20% on SPECint2006, but no observable effect on the Firefox browser).

- Finally, we present **Exterminator**, a runtime system that automatically isolates and corrects buffer overflows and dangling pointers. Using information gathered over multiple program runs, it produces *runtime patches* that correct and prevent similar errors from manifesting in future executions. These patches are *semantically sound*, meaning that they cannot introduce further errors into otherwise correct programs. Exterminator has several different modes of operation for both debugging and deployment scenarios. Its error isolation algorithms are statistically sound and have and provably low false positive and negative rates.

While DieHarder and Archipelago enhance security by lowering the probability of a single attack's success, Exterminator adapts the program execution when errors occur, making any vulnerability a moving target. After repeated failed attacks, Exterminator will detect and isolate the error, eliminating or changing the vulnerability. This combination of adaptation and randomization makes most attacks infeasible.

Outline

This thesis is organized as follows. In Chapter 1, we describe several types of memory errors and explain their impact on reliability and security. We also present background material on dynamic memory allocation, including information on state-of-the-art general-purpose memory allocators. We then describe the hardware and operating system mechanisms for virtual memory support upon which Archipelago and DieHarder depend. In Chapter 2, we present a threat model for heap attacks and describe several attack strategies that target allocator weaknesses. In the subsequent chapters, we describe three runtime systems that enhance reliability and security. In Chapter 3, we present Archipelago and demonstrate that it greatly increases error tolerance and security over DieHard for server applications. In Chapter 4, we describe and analyze DieHarder, an allocator which combines the best properties of Archipelago and DieHard to provide maximal protection against security vulnerabilities while remaining practical for general-purpose programs. Finally, in Chapter 5, we introduce Exterminator, and demonstrate its effectiveness at automatically isolating and correcting buffer overflow and dangling pointer errors. In Chapter 6, we provide an overview of previous work, including work specifically targeting memory errors as well as general software fault tolerance and error avoidance. Finally, we summarize our contributions and conclude in Chapter 7.

CHAPTER 1

BACKGROUND

In this thesis, we address two types of memory errors in the context of dynamic memory management: buffer overflows and dangling pointers. In this chapter, we first introduce dynamic memory management and discuss the major design choices in that space. We then discuss the properties of each type of memory error.

1.1 Dynamic Memory Allocation

The functions that support memory management for C and C++ (`malloc` and `free`, `new` and `delete`) are implemented in the C runtime library. Different operating systems and platforms implement these functions differently, with varying design decisions and features. In nearly all cases, the algorithms underpinning these allocators were primarily designed to provide rapid allocation and deallocation while maintaining low fragmentation [82], without any thought given to their interaction with memory errors. We describe the allocation algorithms used by Windows, Linux, FreeBSD, and OpenBSD, focusing on implementation details with reliability and security implications. Table 1.1 summarizes the security-related characteristics of these allocators.

1.1.1 Freelist-based Allocators

The memory managers used by both Windows and Linux are **freelist-based**: they manage freed space on linked lists, generally organized into bins corresponding to a range of object sizes. Figure 1.1 illustrates an allocated object within the Lea

	Windows	DLMalloc 2.7	PHKmalloc	OpenBSD
No freelists (§ 1.1.1)			✓	✓
No headers (§ 1.1.1)			✓	✓
BiBOP (§ 1.1.2)			✓	✓
Fully-segregated metadata (§ 1.1.2.1)				✓
Destroy-on-free (§ 1.1.2.1)				✓?
Sparse page layout (§ 1.1.2.1)				✓
Placement entropy (bits) (§ 1.1.2.1)	0	0	0	4
Reuse entropy (bits) (§ 1.1.2.1)	0	0	0	5.4

Table 1.1. Allocator security properties (see the appropriate section for explanations). A check indicates the presence of a security-improving feature; a question mark indicates it is optional. While OpenBSD's allocator employs a range of security features, the systems described in this thesis provide significantly higher reliability and security.

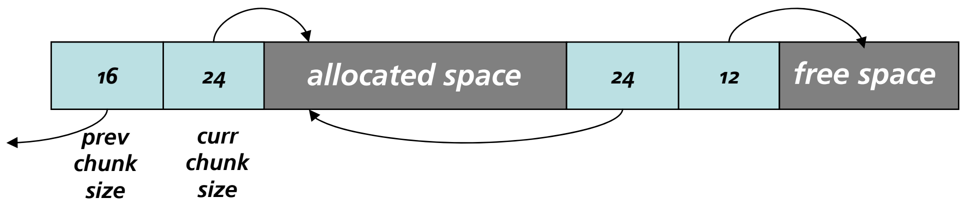


Figure 1.1. A fragment of a freelist-based heap, as used by Linux and Windows. Object headers precede each object, which make it easy to free and coalesce objects but allow overflows to corrupt the heap.

allocator (DLmalloc). Version 2.7 of the Lea allocator forms the basis of the allocator in GNU libc [44].

Inline metadata. Like most freelist-based allocators, the Lea allocator prepends a **header** to each allocated object that contains its size and the size of the previous object. This metadata allows it to efficiently place freed objects on the appropriate free list (since these are organized by size), and to *coalesce* adjacent freed objects into a larger chunk.

In addition, freelist-based allocators typically thread the freelist through the freed chunks in the heap. Freed chunks thus contain the size information (in the headers) as well as pointers to the next and previous free chunks on the appropriate freelist (inside the freed space itself). This implementation has the significant advantage over

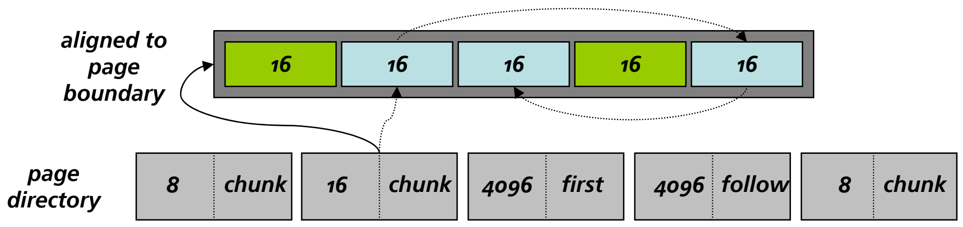


Figure 1.2. A fragment of a segregated-fits BiBOP-style heap, as used by the BSD allocators (PHKmalloc and OpenBSD). Memory is allocated from page-aligned chunks, and metadata (size, type of chunk) is maintained in a page directory. The dotted lines indicate the list of free objects inside the chunk.

external freelists of requiring no additional memory to manage the linked list of free chunks.

Unfortunately, inline metadata also provides an excellent attack surface. Even small overflows from application objects are likely to overwrite and corrupt allocator metadata. This metadata is present in all applications, allowing application-agnostic attacks techniques. Attackers have found numerous ways of exploiting this inherent weakness of freelist-based allocators, including the ability to perform arbitrary code execution (see Section 2.2 for attacks on freelist-based allocators, and Section 2.3 for countermeasures).

1.1.2 BiBOP-style Allocators

In contrast to Windows and Linux, FreeBSD's PHKmalloc [40] and OpenBSD's current allocator (derived from PHKmalloc) employ a heap organization known as *segregated-fits BiBOP-style*. Figure 1.2 provides a pictorial representation of part of such a heap. The allocator divides memory into contiguous areas 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" [33]. BiBOP allocators were originally used to provide cheap access to type data for high-level languages, but they are also suitable for general-purpose allocation.

In addition to dividing the heap into pages, both PHKmalloc and OpenBSD's allocator ensure that all objects in the same page have the same size—in other words, objects of different sizes are *segregated* from each other. The allocator stores object size and other information in metadata structures either placed at the start of each page (for small size classes), or allocated from the heap itself. A pointer to this structure is stored in the page directory, an array of pointers to each managed page. The allocator can locate the metadata for individual pages in constant time by masking off the low-order bits and computing an index into the page directory.

On allocation, PHKmalloc first finds a page containing an appropriately sized free chunk. It maintains a list of non-full pages within each size class. These freelists are threaded through the corresponding page metadata structures. Upon finding a page with an empty chunk, it scans the page's bitmap to find the first available free chunk, marks it as allocated, and returns its address.

Page-resident metadata. As opposed to freelist-based heaps, BiBOP-style allocators generally have no inline metadata: they maintain no internal state between allocated objects or within freed objects. However, they often store heap metadata at the start of pages, or within metadata structures allocated adjacent to application objects. This property can be exploited to allow arbitrary code execution when a vulnerable application object adjacent to heap metadata can be overflowed [5] (see Section 2.2.1).

1.1.2.1 OpenBSD Allocator

OpenBSD originally used PHKmalloc, but recent versions of OpenBSD (since version 4.4, released in 2008) incorporate a new allocator based on PHKmalloc but heavily modified to increase security [51]. It employs the following techniques:

- **Fully-segregated metadata.** OpenBSD's allocator maintains its heap metadata in a region completely separate from the heap data itself, so overflows from application objects cannot corrupt heap metadata.

- **Sparse page layout.** The allocator allocates objects on pages provided by a randomized `mmap` which spreads pages across the address space. This sparse page layout effectively places unmapped "guard pages" between application data, limiting the exploitability of overflows.

- **Destroy-on-free.** Optionally, OpenBSD's allocator can scramble the contents of freed objects to decrease the exploitability of dangling pointer errors.

- **Randomized placement.** Object placement within a page is randomized by a limited amount: each object is placed randomly in one of the first 16 free chunks on the page.

- **Randomized reuse.** The allocator delays reuse of freed objects using a randomly-probed delay buffer. The buffer consists of 16 entries, and on each `free`, a pointer is stored into a random index in this buffer. Any pointer already occupying that index is then actually freed.

Together, these modifications dramatically increase security, although the randomized placement and reuse algorithms are of limited value. We discuss these limitations further in Sections 2.2.1.3 and 2.2.3.1.

1.2 Memory Errors

We specifically target two types of memory errors prevalent in C and C++ applications: buffer overflows and dangling pointers. In this section, we describe the causes, effects, and consequences of these errors.

```c
char arr[40];
char* ptr = arr;
ptr[50] = 'a';
```

(a) Stack overflow

```c
char* ptr =
  (char*) malloc (40);
ptr[50] = 'a';
```

(b) Heap overflow

Figure 1.3. Buffer overflows from stack and heap sources

1.2.1 Buffer Overflows

A *buffer overflow*, or *out-of-bounds write*, occurs when the program writes through a pointer to a location outside the correct memory block. The memory block may be dynamically allocated via `malloc` or automatically allocated on the stack. These errors often occur when writing to an array using an index that exceeds the size of the array.

The program starts with a *base pointer*, which is the address acquired from `malloc` or taking the address of a stack-allocated block. Through some pointer arithmetic, it produces an invalid *derived pointer* and writes through it. For example, Figure 1.3(a) shows an overflow through a stack-derived pointer, i.e., the written address is constructed from a pointer to a memory block on the stack. Because the index used to compute the address of the write (50) is greater than the size of the memory block (40), an overflow occurs.

The program may perform an out-of-bounds write to both stack- and heap-derived pointers. The consequences of the overflow depend on the location of the base memory block. In the case of a stack-derived pointer, the program may overwrite data in another stack-resident memory block or metadata such as the return address or the previous stack pointer. In the latter case, an attacker can overwrite the return address and cause the program to jump to an arbitrary instruction. This type of attack is often referred to as a *stack smash* and remains a common cause of remote security vulnerabilities.

Alternatively, the program may write through an out-of-bounds heap-derived pointer. In this case, the overwritten location may be another heap-resident memory block or metadata used by the dynamic allocator. Corrupting the allocator metadata almost always results in crashes. Other data commonly stored on the heap, such as function pointers and C++ vtable addresses, can be overwritten by attackers, resulting in security vulnerabilities similar to stack-based overflows. Conover et al. describes early techniques for exploiting heap overflows [20]. Huku describes the state-of-the-art for GNU libc (DLmalloc) [35], while McDonald and Valasek provide a detailed overview of attack strategies for the Windows XP heap [48].

1.2.2 Dangling Pointers

A *dangling pointer* is caused by the program accessing a memory location derived from a previously-freed pointer to a heap-resident memory block. These errors are benign until the allocator recycles the memory block for a subsequent `malloc` request. In this case, a memory access through the dangling pointer will read or write data in a different object. A read will return unexpected incorrect data, while a write will overwrite data in the new object, causing data corruption.

Figure 1.4 shows an idealized dangling pointer error. The intent is that `ptr1` and `ptr2` point to different strings. However, due to the dangling pointer error, the subsequent allocation of `ptr2` aliases `ptr1` under many allocators (those which use LIFO freelists). Thus the modification of `ptr2`'s buffer corrupts `ptr1`'s data.

1.3 Virtual Memory

Two of the systems introduced in this thesis make extensive use of operating system support for virtual memory management. This section defines some important terms and concepts and describes the virtual memory primitives used by these systems.

```
char* ptr1 = (char*)malloc(40);
char* ptr2 = 0;

strncpy(ptr1,"The␣quick␣brown␣fox",40);

// Prematurely free ptr1
free(ptr1);

ptr2 = (char*)malloc(40);
strncpy(ptr2,"jumped␣over",40);

// ptr1 and ptr2 are the same string
```

Figure 1.4. Dangling pointer error with corruption

1.3.1 Overview

Virtual memory is a level of indirection between programs and the hardware's underlying physical memory. Programs have a large, fixed-size *address space* of *virtual pages*, some of which are backed by physical *page frames*. The operating system transparently controls which virtual pages are backed by which page frames.

The OS can *evict* contents of stale pages to disk and reclaim their physical page frames for use by other processes. When the program does access an evicted page, it incurs a *page fault*, a hardware trap that the OS handles by transparently copying the contents back into some page frame. It may page out some other page in order to free up a frame. The OS performs paging transparently to user applications.

The original intent of virtual memory was to provide support for *paging*. Most programs obey the *working set hypothesis*, that is, the set of pages accessed during some time window is smaller than the total amount of consumed virtual memory [24]. Contents of pages not within this working set (stale pages) are not immediately needed, and thus can remain evicted without affecting the program's performance.

By exploiting the working set hypothesis, virtual memory allows multiprogrammed systems to run more simultaneous applications than would otherwise fit in physical

RAM. As long as the combined working sets consume less than the amount of physical memory in the machine, few page faults occur, and performance is good. However, if physical memory is too small, *thrashing* may occur, where freshly-evicted pages are accessed quickly, causing continuous page faults. During thrashing, performance slows down by several orders of magnitude, as the system is constantly moving data back and forth from disk rather than doing useful work.

Aside from paging, virtual memory is useful to provide *memory protection* in two senses. First, different processes have separate virtual address spaces, into which the physical memory for other processes is not mapped. This prevents processes from reading or writing to memory in other processes, providing isolation. Second, virtual pages have associated *protection modes* which allow pages to be read- or write-protected. Traditionally, operating systems use protection modes to prevent user-mode reads and writes to kernel pages mapped into process address space.

1.3.2 Core Programming Interface

The underlying hardware determines the parameters of the virtual memory environment presented to the OS and to user processes. It fixes the size of the *virtual address space*, the region of addressable memory. 32-bit architectures such as x86 provide a 4 GB (2^{32}) virtual address space, though OS limitations generally limit the amount of this space usable by user processes to 2–3 GB. 64-bit architectures theoretically allow 2^{64} bytes of addressable memory, but current architectures limit this to less, e.g. 2^{48} bytes on current x86-64 systems.

On these architectures, virtual address space is divided into 4 kB pages. 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 signals a segmentation fault to the process, usually resulting in a program crash.

Processes map virtual address space via the `mmap` system call (`VirtualAlloc` on Windows). This operation *reserves* the address range, so that subsequent operations cannot allocate overlapping memory. Reserved pages do not necessarily have associated physical page frames.

When a process touches a reserved page for the first time, the OS transparently maps a physical page frame, thus *committing* the page. The kernel may acquire the frame by evicting another page, and so the kernel fills the contents of the frame with zeroes before mapping it to the new virtual page. Once committed, subsequent accesses to the page do not result in page faults until the kernel evicts the page.

Mapped virtual pages may be returned to the OS using the `munmap` system call (`VirtualFree` on Windows). This call removes the the mapping to a physical frame, if any, and returns the virtual page to the unmapped state.

1.3.3 Implementation Details

Virtual memory is an extra level of indirection, and thus requires a map between virtual pages and physical frames, called the *page table*. On most architectures, the page table is implemented as a multilevel structure. For example, on x86, the first 10 bits of address are used to index into the root of the structure and find a page table block (PTB). The next 10 bits index into the PTB, resulting in a *page table entry* which contains the physical address of the corresponding page frame or the location of the page's contents on disk.

Because every read or write operation to memory requires consulting the page table to establish the true physical address, the processor caches commonly-used mappings in a cache, called the *translation lookaside buffer*, or TLB. TLBs vary in size, from small, 128-entry direct-mapped caches on x86 microarchitectures before 2009, to larger, multi-level caches seen on IBM Power systems and the newer Intel

Nehalem architecture. The size of the virtual memory range cacheable by the TLB is referred to as *TLB reach*.

1.3.4 Extensions

While `mmap` and `munmap` provide essential support for virtual memory support, most operating systems provide additional functionality. The `mprotect` system call allows user programs to modify the protection bits on mapped virtual pages. For example, a user program could disallow execution for pages in its stack and heap segments, effectively preventing a range of security vulnerabilities, such as class stack smashing. Another use is write-protecting pages in the mature space of a generational garbage collector to implement a cheap hardware page-marking write barrier [38].

Archipelago relies on read protection to implement its user-mode page compression.

In addition to `mmap` and `mprotect`, many operating systems provide other APIs that provide hints about memory usage in order to improve performance of the virtual memory system. For example, a program can use the `mlock` system call to instruct the kernel to never page out a given page range.

Most important to our systems is the `madvise` system call. An application can invoke `madvise(MADV_FREE)`[1] to inform the kernel that the data on a range of pages is no longer needed, and it can therefore discard the physical frame rather than writing it to disk. In contrast to the `munmap` system call, `madvise(MADV_FREE)` does not unmap the virtual page, but rather changes its state from committed to reserved. 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.

[1]POSIX specifies the described semantics for MADV_FREE. Linux provides different semantics, where MADV_DONTNEED provides the functionality described here.

1.4 Probabilistic Memory Safety

This section describes *probabilistic memory safety*, a strategy for providing protection against buffer overflows. Systems implementing probabilistic memory safety approximate an *infinite heap memory manager*. We describe both concepts in this section.

1.4.1 Infinite Heap Semantics

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 using an infinite-sized, fresh block of memory (a *boundless memory block* [63]), and are never deallocated.

Portable, correct C programs cannot distinguish between an infinite heap memory manager and a normal allocator because the semantics of the heap remain unchanged. However, an infinite heap grants additional semantics to memory errors (which cause undefined behavior according to the original semantics). 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. Thus, a program containing memory errors would execute correctly under the enhanced semantics, as long as it does not contain uninitialized reads.

1.4.2 Probabilistic Approximation

Real runtime systems cannot implement infinite heap semantics since real hardware has finite resources. Systems can approximate infinite semantics by using an M-*heap*—a heap that is M times larger than needed. By placing objects uniformly randomly across an M-heap, the expected separation between any two objects of size N is $M - 1$ times the size of an object. Small overflows thus become benign (do not overwrite useful data) with high probability. By *randomizing* the choice of freed objects to reuse, the system minimizes the likelihood of reallocating recently freed objects and overwriting their contents, decreasing the chance that a dangling pointer error results in corruption. This heap thus provides *probabilistic memory safety*, a quantifiable probabilistic guarantee that memory errors that occur in the program remain benign.

1.4.3 DieHard

DieHard [9] was the first system to implement probabilistic memory safety. DieHard is a bitmap-based, fully-randomized allocator. The latest version of DieHard, upon which Exterminator (Chapter 5) is based, adaptively sizes its heap to be M times larger than the maximum needed by the application [10] (see Figure 5.2). This version of DieHard allocates memory from increasingly large chunks called *miniheaps*. Each miniheap contains objects of exactly one size. If an allocation would cause the total number of objects to exceed $1/M$, DieHard allocates a new miniheap that is twice as large as the largest existing miniheap.

During allocation, DieHard first selects a miniheap by choosing one randomly from the proper size class, weighted by miniheap size. After selecting a miniheap, DieHard randomly probes its bitmap for a free bit. If the bit is set, it starts over, repeating the random miniheap selection. Allocation takes $O(1)$ expected time. Freeing a valid object resets the appropriate bit. DieHard's use of randomization across an over-provisioned heap makes it probabilistically likely that buffer overflows will land on free space, and unlikely that a recently-freed object will be reused soon, making dangling pointer errors rare.

In a DieHard M-heap, the likelihood of no live objects being overwritten by an overflow N objects in size is $(1 - \frac{1}{M})^N$ [9]. The probability of the data referenced by a dangling pointer being intact after A allocations is at least $1 - \left(\frac{A}{F}\right)$, where F is the number of free object slots in the corresponding size class.

DieHard optionally uses replication to increase the probability of successful execution. In this mode, it broadcasts inputs to a number of replicas of the application process, each equipped with a different random seed. A voter intercepts and compares outputs across the replicas, and only actually generates output agreed on by a plurality of the replicas. The independent randomization of each replica's heap makes the probabilities of memory errors independent. Replication thus exponentially decreases the likelihood of a memory error affecting output, since the probability of an error striking a majority of the replicas is low.

1.5 Discussion

We focus on explicitly-managed environments, where memory errors have been traditionally undetected by the runtime system and result in program crashes, security vulnerabilities, and silent data corruption. Managed environments provide better safety guarantees: some errors are impossible, such as dangling pointers, and the runtime system can detect other errors, like overflows, preventing data corruption. However, some errors, particularly memory leaks, still occur in managed systems, and have been the focus of recent work [17].

CHAPTER 2
MEMORY ERROR EXPLOITATION

In this chapter, we show how interactions between memory errors and the memory allocator create opportunities for attackers to consistently exploit errors. We first present a threat model and discuss the environment in which attacks occur. We then discuss attack techniques that exploit specific interactions between memory errors and allocator behavior. Finally, we discuss specific countermeasures added to existing allocators that harden them against some of these attack strategies.

2.1 Threat Model

This section characterizes the landscape for heap-based attacks and presents our threat model.

2.1.1 Landscape

The power of potential heap attacks is affected by several factors, including the presence of memory errors, the kind of application being attacked, and whether the attacker has the ability to launch repeated attacks.

Presence of memory errors. The first and most important factor is the existence of a memory error, and the attacker's ability to trigger the code path leading to the error. A program with no memory errors is not vulnerable to heap-based attacks. Even mature programs such as Firefox or Adobe Acrobat have latent memory errors that can be triggered by unexpected inputs and allow exploitation.

Application class. The kind of application under attack affects the attacker's ability to control heap operations. Many attacks assume an unfragmented heap, where the effects of heap operations are predictable. For example, when there are no holes between existing objects, new objects will be allocated contiguously on a fresh page. Many attack strategies assume the ability to allocate enough objects to force the heap into a predictable state before launching the actual attack.

When attacking a web browser, the attacker can run scripts written in JavaScript or Flash. In most current browsers, JavaScript objects are allocated in the same heap as the internal browser data, allowing the attacker to control the state of the application heap. Sotirov describes a sophisticated technique called *Heap Feng Shui* that allows attacks on browsers running JavaScript to ensure predictable heap behavior [75].

Server applications are generally less cooperative. The number and types of allocated objects can be fixed by the application. However, an attacker may be able to place the heap into a predictable state by issuing concurrent requests, forcing the application to allocate a large number of contemporaneously live objects.

Other applications may provide attackers with no ability to cause multiple object allocations. For example, many local exploits target programs that run with enhanced privileges (e.g. setuid root) which may run for a short time and then terminate. In many cases, the attacker is limited to controlling the command-line arguments and the resulting heap layout.

Ability to launch repeated attacks. An application's context defines the attacker's ability to repeatedly launch attacks. In a web browser, if the first attempt fails and causes the browser to crash, the user may not attempt to reload the page. In this case, the attack has only one chance to succeed per target. On the other hand, server applications generally restart after crashes to ensure availability, providing the attacker with more opportunities. If the server assumes an attack is in progress and does not restart, then the vulnerability becomes a denial of service.

Given enough time, an attacker with any probability of success will eventually succeed. However, if the allocator can decrease this probability, the system maintainer may be able to analyze the attack and fix the application error before the attacker succeeds.

Randomization techniques such as address-space layout randomization (ASLR) are designed to provide such unpredictability. For example, Shacham et al. showed that ASLR on 32-bit systems provides 16 bits of entropy for library address and can thus be circumvented after about 216 seconds [69]. On 64-bit systems providing 32 bits of entropy, however, the attack would require an expected 163 days. During this time, it would be feasible to fix the underlying error and redeploy the system.

While one can imagine a hypothetical supervisor program that detects incoming attacks, such a system would be hard to make practical. While it could detect a series of crashes coming from a single source, sophisticated attackers control large, distributed networks which allow them to coordinate large numbers of attack requests from different sources. Shacham et al. discuss the limitations of such systems in more detail [69].

However, more sophisticated techniques can limit the vulnerability of systems to repeated attacks. Systems such as Rx [59], Exterminator (Chapter 5), and ClearView [58] can detect heap errors and adapt the application to cope with them. For example, Exterminator can infer the size of an overflow and pad subsequent allocations to ensure that an overflow of the same size does not overwrite data.

The threat model. We assume the attacker has the power to launch repeated attacks and allocate and free objects at will. Repeated attacks are most useful against Internet servers, while the unlimited ability to allocate and free objects is most useful against web browsers (especially when executing JavaScript). This model thus assumes the worst-case for prime attack targets in the real world.

We analyze vulnerabilities based on a single exploit attempt. The lower the likelihood of success of a single attack, the longer the expected time before the application is compromised. Given enough time, the error can be corrected manually, or a system like Exterminator can adapt the application to correct it.

2.2 Attacks

We now explain in detail how heap-based exploits work, and how these interact with the underlying heap implementations. Exploits often directly exploit heap-based overflows or dangling pointer errors (including double frees), but can also start with **heap spraying** attacks [30] and then later exploit a vulnerability.

We abstract out each of these attacks into an **attack model**. We illustrate these models with examples from the security literature, and show how particular memory management design decisions facilitate or complicate these attacks.

2.2.1 Heap Overflow Attacks

Perhaps the most common heap attack strategy exploits an overflow of an object adjacent to heap metadata or application data.

2.2.1.1 Overflow attack model

Abstractly, an overflow attack involves two regions of memory, one **source chunk** and one or more **target chunks**. Target chunks can include application data or heap metadata, including allocator freelist pointers. The attacker's goal is to overwrite some part of target chunk with attacker-controlled data.

A real attack's success or failure depends on application behavior. For example, an attack overwriting virtual function table pointers only succeeds if the application performs a virtual call on a corrupted object. However, details of such application behavior is outside the scope of our attack model, which focuses only on the interaction between the heap allocator and overflows. For purposes of analysis, we are pessimistic from the defender's viewpoint: **we assume that an attack succeeds whenever a target chunk is overwritten**.

Note that the attacker's ability to exploit a heap overflow depends on the specific application error, which may allow more or less restricted overflows. For example, off-by-one errors caused by failure to consider a null string termination byte allow only the overflow of 1 byte, with a specific value. In general, `strcpy`-based attacks do not allow the attacker to write null bytes. On the other hand, some errors allow overwrites of arbitrary size and content.

2.2.1.2 Specific attacks

An overflow attack may target either heap metadata or application data. In some cases, a single, specific heap object may be the target, such as a string containing a filename. In others, there may be many targeted chunks. For example, a potential target for application data attacks is the virtual function table pointer in the first word of C++ objects with virtual functions. In some applications, many objects on the heap have these pointers and thus are viable targets for attack. Other attacks target inline heap metadata, present in the first words of every free chunk.

Early attacks. The earliest heap overflow attacks targeted application data such as filename buffers and function pointers [20]. A susceptible program allocates two objects, the source (overflowed) chunk and an object containing a function pointer (the target chunk). A successful attack forces the allocator to allocate the source chunk and victim chunk contiguously. It then overflows the buffer, overwriting the function pointer with an attacker-controlled address. If the chunks are not adjacent, a more general attack may overwrite multiple objects in between the buffer and the vulnerable object.

Freelist metadata attacks. Solar Designer first described an attack relying on specifics of the heap implementation [74]. The attack applies to any allocator that embeds freelist pointers directly in freed chunks, such as DLmalloc and Windows. The specific attack described allowed a hostile web server to send a corrupt JPEG image allowing arbitrary code execution within the Netscape browser.

This attack overwrites words in the free chunk header, overwriting the freelist pointers with a specific pointer (generally to shellcode) and the address of a target location. Candidate target locations include function pointers in heap metadata structures, such as `__free_hook` in DLmalloc, which is called during each `free` operation. When the corrupted free chunk is reallocated, the allocator writes the pointer to the target location.

In the worst case for this attack, every free chunk is a target. Once a free chunk is corrupted, the attacker can simply force allocations until the chunk is reused. However, existing attacks in the literature target a single, attacker-controlled free chunk.

Other metadata attacks. BBP describes overflow attacks targeting PHKmalloc metadata, which resides at the beginning of some pages and also allocated within the heap itself [5]. In this case, the attacker does not directly control the target chunks. However, he may be able to indirectly force the allocation of a metadata object by allocating a page worth of objects of certain size classes. Target chunks include these page info structures.

2.2.1.3 Allocator Analysis

A number of allocator features have a direct impact on their vulnerability to overflow attacks.

Inline metadata. Allocators such as DLmalloc and Windows that use inline metadata inherently provide many target chunks. For some attacks, effectively any chunk in the heap could be overwritten to cause a remote exploit. For example, a patient attacker relying on freelist operations (an "`unlink` attack") could overwrite the freelist pointers in an arbitrary free chunk, then simply wait for that chunk to be reused. These allocators are similarly vulnerable to other such attacks, such as those targeting an object's size field.

Page-resident metadata. Allocators with no inline metadata, such as PHKmalloc, may still have allocator metadata adjacent to heap objects. PHKmalloc places page info structures at the beginning of some pages (those containing small objects), and allocates others from the heap itself, in between application objects. Those allocated from the heap itself are obviously vulnerable to overwrites, especially if the attacker can control where they are allocated due to determinism in object placement. PHKmalloc also lacks guard pages, meaning that the page info structures placed at the beginning of pages may also be adjacent to overflowable application chunks.

Guard pages. Guard pages can protect against overflows in multiple ways. First, for allocators like PHKmalloc which place metadata at the beginning of some pages, guard pages could be used to protect that metadata against overflows (though they are not). Deterministically placing a guard page before each page with metadata provides protection against contiguous overruns (the most common case), but not against underruns or non-contiguous overflows (such as an off-by-one on a multidimensional array).

Second, guard pages provide gaps in memory that cannot be spanned by contiguous overflow attacks, limiting the number of heap chunks that can be overwritten by a single attack. In this sense, guard pages protect application data itself. However, if the allocator is sufficiently deterministic, an attacker may be able to ensure the placement of the source chunk well before any guard page, allowing an attack to overwrite many chunks.

Canaries. The use of canaries to protect heap metadata an application data may protect against overflows in some cases. However, their effectiveness is limited by how often the canaries are checked. Metadata canaries may be checked during every heap operation and can substantially protect metadata against overflows. However, allocators that place canaries between heap objects must trade off runtime efficiency for protection. For example, an overflow targeting a function pointer in application data requires no heap operations: only the overwrite and a jump through the pointer. Since allocators that check canaries only do so on `malloc` and `free`, they cannot protect against all such attacks.

Randomized placement. All existing allocators that do not explicitly randomize object placement can be forced to allocate contiguous objects, assuming enough control of allocations and frees by the attacker. Techniques such as Heap Feng Shui are designed to force the allocator into such a deterministic state in order to enable reliable exploitation of vulnerabilities.

OpenBSD randomizes placement of heap objects to a limited extent. This approach reduces the reliability of overflow exploits by randomizing which heap chunks are overwritten by any single overflow. Attacks that depend on contiguous objects are also complicated, since it is unlikely that any given objects will be contiguous in memory.

However, overflow attacks able to span multiple heap chunks need not rely on contiguously-allocated objects. As long as the target object is placed after the source object on the same page, the attacker can overwrite the target. The extent of placement randomization affects the probability of such an attack's success.

OpenBSD's limited randomization allows certain such attacks to succeed with high probability. In an unfragmented heap, successive allocations of the same size objects will be clustered on the same page, even though their placement is randomized within that page. An attacker that can control heap operations so that the source and target are allocated on the same page has a 50% probability of allocating the source at a lower address than the target, enabling the attack to succeed.

For small objects, object placement is not fully randomized within a page because the allocator uses only 4 bits of entropy for a single allocation. For example, two successive allocations on a fresh page will always lie within 16 chunks of each other. An attack can exploit this property to increase attack reliability by limiting the length of the overflow, reducing the risk of writing past the end of a page and causing the application to crash.

2.2.2 Heap Spraying Attacks

Heap spraying attacks are used to make exploitation of other vulnerabilities simpler. In modern systems, guessing the location of heap-allocated shellcode or the address of a specific function for a return-to-libc attack can be difficult due to ASLR. However, on many systems, the heap lies within a restricted address space. For example, on 32-bit systems the heap generally lies within the first 2 GB of virtual address space. If the attacker allocates hundreds of megabytes of shellcode, jumping to a random address within this 2 GB region has a high probability of success.

2.2.2.1 Heap spraying attack model

To successfully exploit a heap spray, the attacker must guess the address contained within some (large) set of attacker-allocated objects. However, the attacker need not guess a pointer out of thin air. The simplest attack exploits an overflow to overwrite an application pointer with the guessed value. However, if this pointer already references the heap, overwriting only the low-order bytes of the pointer on a little-endian machine results in a different pointer, but to an address close to the original address. An attacker often knows the address of a valid heap object and can use this knowledge to guess the address of a sprayed object. This knowledge may be acquired either implicitly due to a partial overwrite, or explicitly based on information leakage.

To account for these effects, we consider two heap spraying attack models. Both require the attacker to guess the address of one of a specific set of sprayed objects, V. The models differ in the information known to the attacker:

- **No *a priori* knowledge.** In the first model, the attacker must guess an address with no *a priori* knowledge of valid heap addresses.
- **Known address attacks.** In the second attack model, the attacker knows the address of a valid heap object. In some cases, the attacker may control when the known object is allocated. For example, allocating it temporally between two shellcode buffers makes it easy to guess a shellcode address if the heap allocates objects contiguously. This model is more general and significantly stronger than the ability to partially overwrite a pointer value. In the latter case, the attacker does not know an exact address, and can only guess addresses within 256 or 64K bytes (when overwriting 1 or 2 bytes, respectively).

2.2.2.2 Allocator Analysis

We quantitatively analyze allocator design choices with respect to heap spraying attacks under both attack models.

No *a priori* knowledge. First, we analyze the probability of an attacker guessing the address of one of a set V of target objects without *a priori* information. V models the set of objects sprayed into the heap. Note that unlike the overflow case, the attacker can cause $|V|$ to be close to $|H|$, the size of the heap. Thus, the probability of guessing the address of heap-allocated shellcode or other target heap data is equivalent to guessing the address of any heap object in the limit.

More formally, we consider the probability $P(a \in V)$, that is, the probability of address a pointing to a valid heap object. Under this model, the attacker knows only the approximate value of $|H|$, the amount of allocated heap memory.

$P(a \in V)$ is almost entirely dependent on the target system's ASLR implementation. For example, on systems without ASLR, an attacker knowing the size of the heap can always guess a valid address. Even with ASLR, an attacker spraying hundreds of megabytes of data into the heap on a 32-bit system has a high probability of guessing the address of a sprayed object. Note that $P(a \in V)$ need not be uniform with respect to a: if the system allocates memory contiguously and $|H| > 2$ GB, then address $a =$ `0x80000000` must contain valid data. However, if the allocator allocates pages randomly and non-contiguously, then the probability need not depend on a itself.

On 64-bit systems, however, the situation is vastly improved. Even on modern x86-64 systems which limit the effective virtual address range to 48 or 52 bits, physical memory limitations restrict the attacker's ability to fill a significant portion of this space. If ASLR randomizes the addresses of the `mmap` region across the entire space, the probability of guessing a valid address is low. Further evaluation of existing ASLR systems, which have been discussed by Shacham et al. [69] and Whitehouse [79], is outside the scope of this dissertation.

Known address attacks. From the allocator's perspective, the problem of guessing the address of an object in V given the address of a heap object $o \in V$ depends upon the correlation of valid addresses with that of o. In most allocators, this correlation is due to contiguous object allocation. The addresses of contiguous objects are dependent upon each other. For example, if the entire heap is contiguous, then the addresses of all heap objects are mutually dependent, and thus amenable to guessing.

Quantitatively, we can evaluate the predictability of object addresses by considering the conditional probability distribution $P(a \in V \mid o \in H)$. An allocator that minimizes this probability *for all* $a \neq o$ is the least predictable.

In a contiguous heap, the address distribution is highly correlated. An address δ bytes after the known object o is valid if and only if o lies within the first $H - \delta$ bytes of the heap. In the worst case, we have no knowledge of the position of o within the heap. The probability of a being a valid address is thus dependent on its distance from o. The validity of the addresses surrounding o are highly correlated.

By contrast, Archipelago allocates each object in a random position on a separate page, compressing cold pages to limit its consumption of physical memory. Since it allocates objects randomly throughout a large address space, Archipelago delivers minimal correlation because all object addresses are independent. The probability $P(a \in V \mid o \in H) \approx P(a \in V)$[1].

Practical allocators must trade off performance with predictability. While Archipelago works well for programs with small heap footprints and allocation rates, it is by no means a general purpose allocator. Practical allocators must allocate multiple objects on the same page in order to provide spatial locality to the virtual memory system. The page granularity thus limits the entropy an allocator can provide, and thus the protection it can supply against heap spray attacks.

2.2.3 Dangling Pointer Attacks

Temporal attacks rely on an application's use of a free chunk of memory. If the use is a write, the error is generally called a dangling pointer error. If the subsequent use is another free, it is called a double-free error. There are two general attack strategies targeting these errors, one based on reuse of the prematurely freed object,

[1]There is still minimal dependence, as Archipelago ensures at most one object is allocated per page.

and another based on a freelist-based allocator's use of free chunks to store heap metadata.

2.2.3.1 Reuse Vulnerabilities

The first strategy exploits the reuse of chunks still referred to by a dangling pointer. The attacker's goal is to change the data contained in the object so that the later (incorrect) use of the first pointer causes an unintended, malicious effect. For example, if the dangled object contains a function pointer, and the attacker can force the allocator to reuse the chunk for an attacker-controlled object, he can overwrite the function pointer. Later use of the original pointer results in a call through the overwritten function pointer, resulting in a jump to an attacker-controlled location. This attack strategy has been described elsewhere [81], but we know of no specific attacks described in the literature.

This strategy exploits the predictability of object reuse by the allocator. A reliable attack can only be created if the attacker knows when the dangled chunk will be recycled. We formalize this by designating the dangled chunk as the **target chunk**. The attacker succeeds by forcing the allocator to recycle the target chunk.

Unlike buffer overflows, where each attempt by the attacker may cause the program to crash (e.g., by attempting to overflow into unmapped memory), repeated attempts to reallocate the dangled chunk need not perform any illegal activity. The attacker just allocates objects and fills them with valid data. This strategy limits the ability the runtime system to cope with such an attack, unless it somehow prevents the original dangling pointer error (e.g., via conservative garbage collection).

To combat reuse-based attacks, an allocator can implement a variety of strategies to delay reuse. First, it can delay reuse for as long as possible, e.g., by using a FIFO freelist. Unfortunately, in a defragmented heap, this policy has little effect.

The allocator could also impose a minimum threshold before objects are recycled. While a fixed threshold would be predictable and thus exploitable, randomized reuse would generally make attacks less reliable. For example, if an attacker has only one chance to force an application to call the overwritten function pointer, randomized object reuse reduces the probability of success.

OpenBSD implements limited reuse randomization by storing freed pointers in a random index of a 16-element array. The object is only actually freed when a subsequent free maps to the same array index. Each subsequent free is thus a Bernoulli trial with a 1/16 probability of success, making the distribution of t, the time before the object is reused, follow a geometric distribution with approximately 5.4 bits of entropy.

2.2.3.2 Allocator Analysis

In this section, we analyze the effect of allocator design on the predictability of object reuse. We evaluate each allocator feature by analyzing the entropy of t, the random variable representing the number of allocations before a just-freed object is recycled.

Freelists. Freelist-based allocators commonly use LIFO freelists. Independent of other allocator features such as coalescing, such freelists always return the most-recently allocated object, providing zero entropy and thus perfect predictability.

BiBOP-style allocators. BiBOP-style allocators may implement different reuse policies. PHKmalloc tracks a freelist of pages, and allocates in address-ordered first-fit within the first page on the freelist. Thus, t depends on the number of free chunks on a page. If the freed object creates the only free chunk on the page, the page was not previously on the freelist, and so the allocator will place it at its head. The subsequent allocation will choose this page, and return the only free chunk, which is the just-freed chunk. An attacker can force this behavior by allocating objects from the same size class as the target in order to eliminate fragmentation before the call to `free`.

Coalescing. Most freelist-based allocators perform coalescing, the merging of adjacent free chunks. When a free chunk is coalesced with an existing free chunk, its size class will change, and thus be placed on an unpredictable freelist. While coalescing is deterministic, it relies on several aspects of the heap layout, making it difficult to create attacks when it occurs. However, in a defragmented heap, the probability of coalescing occurring is low, making it straightforward to work around in existing allocators.

2.2.3.3 Specific Attack: Inline Metadata

The second strategy relies on the behavior of the allocator itself. Freelist-based allocators write metadata into the contents of free chunks. If a dangling pointer points to a free chunk, then it points to overwritten, invalid data. If the attacker can control or predict the data the allocator writes into the freed chunk, he can maliciously corrupt the contents of the object.

Example. Afek describes an exploit that relies on the object layout of C++ objects, combined with the freelist behavior of the Windows heap [1]. On most implementations, the first word of a C++ object with virtual functions contains the pointer to the virtual function table. This same word is also used by freelist-based allocators to store the pointer to the next object on the freelist. Afek's technique allocates a fake vtable object containing pointers to shellcode, then frees the object. Then, the dangling pointer error is triggered, placing the dangled chunk at the head of the freelist and storing a reference to the fake vtable in the first word. When the application erroneously uses the dangled pointer and performs a virtual function call, the runtime looks up the address of the target function from the forged vtable installed by the allocator, resulting in a jump to shellcode.

2.2.3.4 Allocator Analysis

This vulnerability is specific to freelist-based allocators, and does not affect allocators with no inline metadata. BiBOP-style allocators do not write metadata to free chunks, so they cannot be forced to write attacker-controlled data into dangled objects. This vulnerability also exploits deterministic reuse order, discussed in detail in Section 2.2.3.2.

2.3 Countermeasures

Allocator implementors have introduced a variety of techniques to protect inline metadata against attacks. The first countermeasures were freelist integrity checks, included in modern freelist-based allocators to prevent unlink attacks. Instead of naïvely trusting the free chunk header, the allocator ensures that memory pointed to by the heap chunk header is a valid chunk that refers back to the supplied chunk, and thus forms a valid doubly-linked list.

In addition to freelist integrity checks, Windows XP SP2 added an additional countermeasure. Each object header contains a 1-byte cookie computed from a per-heap pseudorandom value and the chunk address. The allocator checks the integrity of this cookie on each `free` operation, (possibly) aborting the program if it fails. An attack that contiguously overflows the previous object must correctly forge this value in order to overwrite freelist pointers. However, some heap metadata, notably the size field, lies before the cookie, allowing small overwrites to modify the inline metadata without corrupting the cookie. McDonald et al. describe a technique that can achieve a single null byte overflow, such as a string terminator [48] (used in the "heap desynchronization" attack described in that work). Furthermore, there are only 256 possible 1-byte values, so if an attack can repeatedly guess random cookies, it will succeed after a relatively low number of trials.

Despite the introduction of these countermeasures, attackers have found new methods of corrupting heap metadata to allow arbitrary code execution. McDonald and Valasek present a comprehensive summary of attacks against the Windows XP Service Pack 2/3 heap [48], and Ferguson provides an accessible overview of techniques targeting DLmalloc [28].

While freelist integrity checks were added to the Windows XP heap in service pack 2, a similar structure called the lookaside list (LAL) was left unprotected, allowing similar attacks. Similarly, the allocator did not consistently check the header cookie (in particular, during LAL operations), making it possible to exploit certain chunk header overwrites without guessing the correct value [2].

More comprehensive protection for chunk headers was added in Windows Vista. In Vista, the entire chunk header is "encrypted" by XORing with a random 32-bit value. All uses of header fields must be decrypted before use, meaning that the allocator *must* consistently check the header integrity in order to function correctly. In order to supply a specific value to a header field, an attacker must determine the 32-bit value, which is harder to brute force than the single-byte cookie.

While header encryption has effectively eliminated the ability of simple buffer overflows to successfully attack heap metadata, the technique is just the most recent reaction to inline metadata attacks. All of these techniques simply cope with an underlying design flaw: allocators with no inline data are not susceptible to this kind of attack.

CHAPTER 3

ARCHIPELAGO: PROVIDING EXTREME BUFFER OVERFLOW PROTECTION

This chapter presents Archipelago, a runtime system that significantly improves the resilience of applications to repeated heap-based memory errors, including large overflows.[1] Archipelago allocates a single object per page, thus treating heap objects as individual islands, surrounded by stretches of unused address space. On modern 64-bit architectures, virtual address space is a plentiful resource. Archipelago trades virtual memory for a high degree of probabilistic memory safety; 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.

In contrast to DieHard, Archipelago provides a much higher degree of probabilistic memory safety. It places a huge amount of virtual memory between objects (several megabytes), ensuring that large overflows do not corrupt other heap data. To limit *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.

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

While this implementation reduces the number of heap pages required, an object-per-page allocator suffers from drastic internal fragmentation, as it effectively rounds each `malloc` request up to a multiple of the page size. To reduce this fragmentation, Archipelago further reduces physical memory consumption by transparently compacting heap objects that are infrequently used. To perform compaction, Archipelago copies the actual object data out of its page into a private heap. The allocator never exposes private heap pointers to the application, keeping it safe from corruption. It then frees the original page, but retains metadata mapping the page address to the location of the object in the private heap. When the application later accesses the object, no physical page frame is mapped to its virtual page, and thus the access causes a page fault. Archipelago transparently handles this fault and uncompacts the object by mapping a fresh page to the existing address and copying the contents back to its original location in virtual memory, allowing the application to again access the object.

Archipelago targets server applications, the class of applications that are most sensitive to memory errors and associated security vulnerabilities. These servers are attractive, high-value targets that are accessible directly from the Internet. We show that Archipelago can provide high levels of safety and reliability for this class of applications. Archipelago allows applications to run even in the face of thousands of memory errors, while keeping performance impact acceptably low. 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 means of protecting certain deployed server applications against heap-based security vulnerabilities.

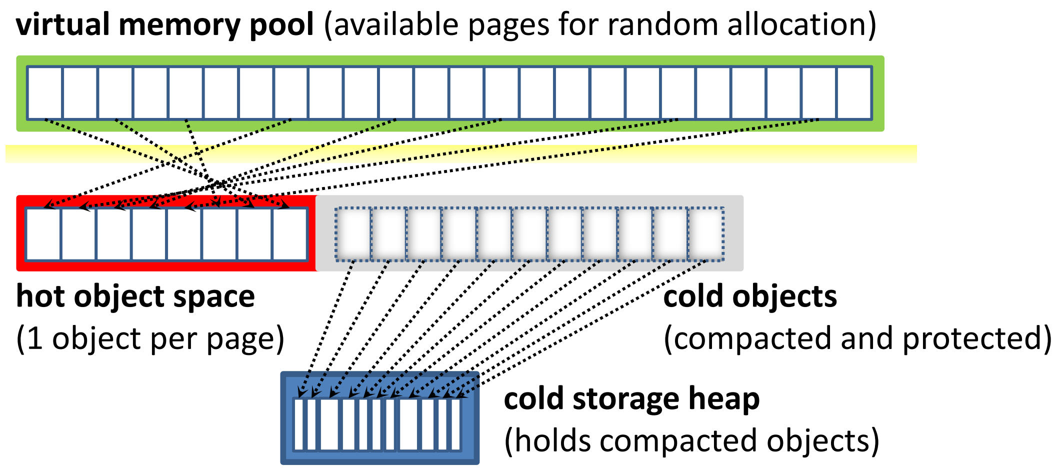

virtual memory pool (available pages for random allocation)
hot object space
(1 object per page)
cold objects
(compacted and protected)
cold storage heap
(holds compacted objects)

Figure 3.1. Archipelago's software architecture. Archipelago randomly allocates heap objects in virtual address space (Section 3.1.1). It tracks the hot objects, which are stored one per page (Section 3.1.2). Cold objects are compacted and placed in cold storage, and the physical memory associated with their page frames is relinquished (Section 3.1.3).

3.1 Archipelago Architecture

Archipelago consists of three modules: 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 3.1 illustrates the architecture. These modules are compiled into a dynamically-linked library that replaces standard memory management routines such as `malloc` and `free` with calls to the Archipelago allocator. Like all systems described in this thesis, Archipelago can be used with existing compiled binaries using a mechanism like Linux `ld.so`'s `LD_PRELOAD`.

3.1.1 Randomizing Object-Per-Page Allocator

Key to Archipelago's protection from memory errors is its object-per-page memory allocator, built using the Heap Layers infrastructure [11]. 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 large, contiguous section of virtual address

```
void * malloc (size_t size) {
        void * page = NULL;
    if (size <= PAGE_SIZE) {
        //object fits on a page
        //obtain random page from the pool
        page = getRandomPage();
    }
    if (page == NULL) {
        //object doesn't fit on the page
        //or pool is full
        //mmap memory directly
        page =
            mmap(roundUpToPageSize(size),
                  MAP_ANONYMOUS);
    }
    if (page == NULL) {
        //mmap failed
        return NULL;
    }
    //add coloring
    void *ptr =
            getRandomColoring(page, size);
    //register page(s) as part
    //of working set
    registerActivePages(page, ptr, size);
    return ptr;
}
```

Figure 3.2. Pseudo-code for Archipelago's `malloc`.

space using `mmap`, and uses this space as a pool from which to draw pages to satisfy allocation requests. Figures 3.2 and 3.3 present pseudo-code for `malloc` and `free`.

In the current implementation, the size of the pool of available pages is a runtime parameter to Archipelago (defaulting to 512 MB) 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.

```
void free (void * ptr) {
    //retrieve size
    size_t size = getObjectSize(ptr);
    //get first page
    void *page = getStartPage(ptr);
    //unregister pages being deleted
    unregisterActivePages(page, ptr, size);
    //discard pages
    //that have been compacted
    discardCompactedPages(page, ptr, size);
    if (size <= PAGE_SIZE) {
        //object fits on page:
        //discard contents
        madvise(page, MADV_FREE);
    } else {
        //object doesn't fit on page:
        //unmap it
        munmap(page,
            roundUpToPageSize(size));
    }
}
```

Figure 3.3. Pseudo-code for Archipelago's `free`.

Allocation: Objects are placed on pages randomly chosen from the pool (Figure 3.2, 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. Otherwise, Linux's prepaging algorithm would normally prefetch some surrounding pages. 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 3.2.4).

Deallocation: When an object smaller than a page in size is deleted, the object-per-page allocator marks the page as free (Figure 3.3, 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 3.2, 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. When an object that was allocated using `mmap` is freed, its memory is immediately released back to the operating system using `munmap` (Figure 3.3, lines 18–19).

3.1.2 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 working set hypothesis. At any given time, the program needs only its working set, a small fraction of its allocated objects.

Archipelago uses a user-mode paging system to evict the mostly-empty pages managed by the object-per-page allocator. Instead or relying on the OS to write the entire contents of the page to disk, it exploits the fact that most space on the page is unused. It compacts unneeded pages in this space to its private heap, avoiding the need for expensive and inefficient disk traffic.

First, the user specifies the desired maximum working set size of the program through an environment variable. If not specified, it defaults to a maximum working set of 5,000 objects. Archipelago compacts cold objects that do not fit within this maximum size. After compaction, it then informs the OS that these now-redundant page frames can be discarded without the need to write them back to disk via `madvise`.

Archipelago tracks all pages containing 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 to the specified maximum working set size. Pages are added to the back of the queue at allocation time. Once the queue becomes full, Archipelago removes and compacts pages from the front of the queue to maintain its maximum size. When a page is uncompacted, it is again added to the back of the queue.

Unlike Archipelago, operating systems generally use LRU approximations such as CLOCK to manage the working set [18]. These algorithms rely on hardware-managed dirty and referenced 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.

Since it does not have access to these bits, Archipelago uses a FIFO policy for managing its working set. Even if these bits were visible to user space and thus exploitable by Archipelago and similar systems, FIFO would likely still be a good choice. FIFO is provably 2-competitive with LRU (that is, it suffers at most twice as many page faults as does LRU on a given workload and memory size) [73]. More sophisticated algorithms would require more overhead for reading referenced bits and tracking page use statistics. Because Archipelago's page fault cost is orders of magnitude lower than when paging to disk, this tradeoff is unlikely to provide better overall performance.

```
 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, hotPages[page]->getDataStart(),
 7          hotPages[page]->getDataSize());
 8      // set trap on future accesses
 9      mprotect(page, PROT_NONE);
10      // mark page as cold
11      coldPages[page] = hotPages[page];
12      hotPages.remove(page);
13      // remember the location of the data
14      coldPages[page].setColdStore(coldStore);
15      // return physical page to OS
16      madvise(page, MADV_FREE);
17  }
```

Figure 3.4. Pseudo-code for Archipelago's compaction routine (Section 3.1.3).

3.1.3 Cold Storage

Archipelago compacts pages not in the current working set into a separate, private heap, called the *cold storage*. This heap stores only compacted objects and uses the general-purpose Lea allocator [44].

When Archipelago compacts a page, it copies the contents of the object into the internal heap (Figure 3.4, lines 2–8). Rather than storing the size of the original object request, Archipelago determines the bounds of the object by scanning from both ends until finding a nonzero word. This effectively increases reliability by copying data written by small overflows, providing a better approximation of infinite heap semantics. This scheme could be easily extended to detect overflows by comparing the size of the used region with the original requested size, though we have not implemented this feature.

After copying the object contents, Archipelago disables direct access to the page by removing read and write access via `mprotect` (line 10). The next time the application tries to access the page, the operating will send a signal to the process, which Archipelago will handle. Finally, Archipelago removes the page from the hot space by calling `madvise` to instruct the virtual memory manager to discard the page contents instead of flushing it to disk (lines 11–18).

Archipelago installs a signal handler that receives segmentation violation signals from the OS. Archipelago uses this signal to restore objects from cold storage on demand (Figure 3.5, lines 28–33). When the handler receives a signal, it first checks whether the access is to a page corresponding to an object in cold storage. If not, then it is a true segmentation violation, and Archipelago terminates the program. However, if the application was trying to access an object in cold storage, the handler "inflates" the object. A naïve implementation of this handler would first unprotect the page and copy the data back from cold storage (Figure 3.5, lines 26–30). However, this implementation is not safe in multithreaded programs, since another thread may read the page after Archipelago unprotects it, but before it finishes copying the data.

Because memory protection is shared across all threads, Archipelago cannot write to the page without unprotecting it for all threads. To avoid this scenario, Archipelago acquires a fresh page at a different address and copies the object contents into it (Figure 3.5, lines 5–11). It then uses `mremap` to install a mapping for this new physical page at correct virtual address (lines 13–14). Only then, once the contents are properly in place, does it unprotect the original virtual page using `mprotect` (line 18).

After this operation, the object is fully transitioned back into hot space, so Archipelago can free the space used to hold the object in cold storage (Figure 3.5, lines 19–23). 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 3.2.5 shows.

```
bool inflate (void *page) {
    // check that page is valid
    if (!coldPages.hasKey(page))
        return false;
    // map a temporary page
    char * newPage = mmap(0, 4096,
        PROT_READ | PROT_WRITE);
    // restore data
    memcpy(newPage + coldPages[page].getOffset(),
        coldPages[page].getColdStore(),
        coldPages[page].getSize());
    // map the new page to the old address
    mremap(newPage, 0, 4096,
        MREMAP_MAYMOVE | MREMAP_FIXED, page);
    // remove the temporary page mapping
    munmap(newPage);
    // enable access to the original page
    mprotect(page, PROT_READ | PROT_WRITE);
    // free the cold space
    coldHeap.free(coldPages[page].getColdStore());
    // mark page as hot
    hotPages[page] = coldPages[page];
    coldPages.remove(page);

    return true;
}

void sigsegv_handler(void *addr) {
    if (!inflate(getPageStart(addr))) {
        // Access outside heap
        abort();
    }
}
```

Figure 3.5. Pseudo-code for Archipelago's uncompaction routine (Section 3.1.3).

3.2 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?

3.2.1 Experimental Methodology

We perform our evaluation on a quiescent dual-socket machine with 8 gigabytes of RAM. Each processor is a 4-core 64-bit Intel Xeon E5345 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 [44], and to DieHard, version 1.1. This version, available from the project website, is an adaptive variant that dynamically grows its heap [10], and so is more space-efficient than the original, published description [9].

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 halves Archipelago's on a memory-intensive microbenchmark, because `madvise` is on Archipelago's normal deallocation path.

Benchmark	**Max live** (bytes)	**Max live** (objects)	**Total memory** (bytes)	**Total objects**	**Alloc rate** (bytes/sec)
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 3.1. Server benchmark characteristics: maximum live size, total allocated memory over the life of the program, and allocation rate.

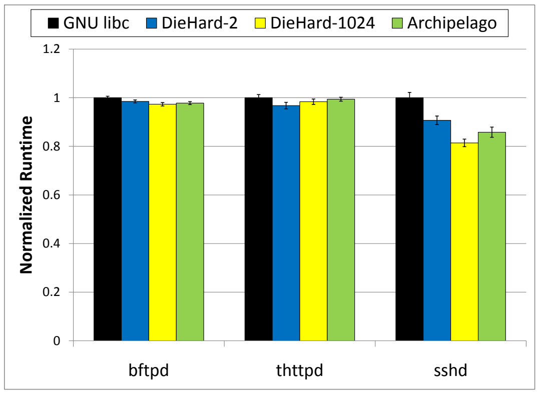
The current Linux kernel (2.6.36) does not contain this patch, though other modifications have been made to the virtual memory system that affect the performance of `madvise`. We have not measured the performance of Archipelago on newer kernels.

3.2.2 Server Application Performance

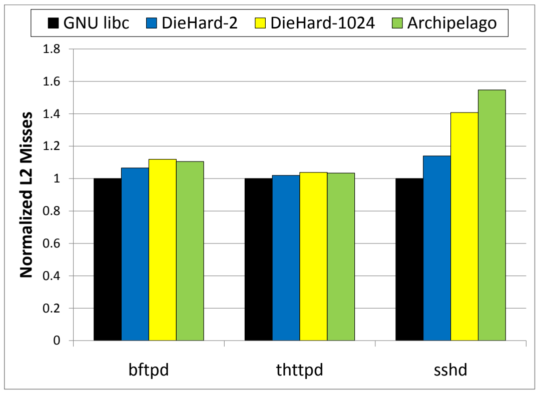
To quantify the performance overhead of using Archipelago, we measure the runtime of a range of server applications running with and without Archipelago. In our experiments, Archipelago uses a memory pool of 512 megabytes, when not otherwise stated. We also compare performance against DieHard with two 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.

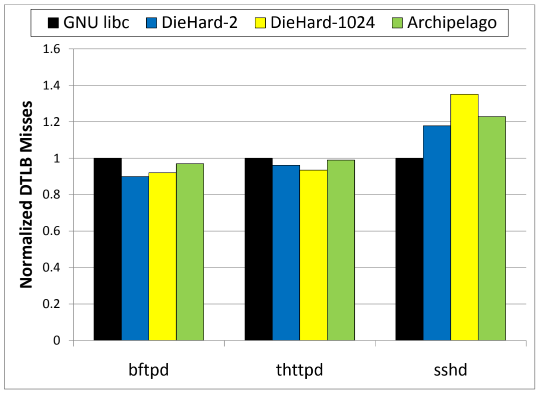
We focus on the CPU performance of our benchmarks by performing all our experiments over the loopback network interface to minimize the performance impact of the network interface. Thus the measured runtime overheads are thus conservative estimates of the performance overhead one would see in practice.



(a) Runtime



(b) L2 Misses



(c) DTLB Misses

Figure 3.6. Performance across a range of server applications (Section 3.2.2), normalized to GNU libc (smaller is better).

Table 3.1 presents the allocation characteristics of the servers in our benchmark suite. Because Archipelago's allocator uses both more CPU time and more memory 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 3.6 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.

3.2.3 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 3.7 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 3.8 shows, Archipelago's ability to control its working set size yields far better performance than

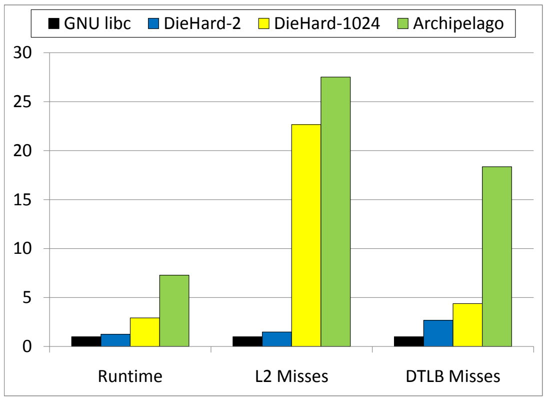


Figure 3.7. Performance metrics for the memory-intensive *espresso* benchmark (Section 3.2.3), normalized to GNU libc (smaller is better).

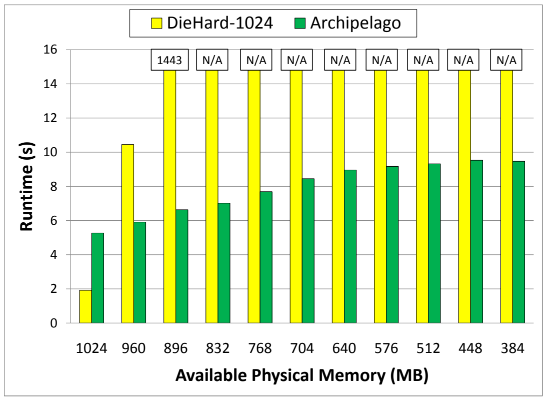


Figure 3.8. Runtime of the memory-intensive *espresso* benchmark under memory pressure (Section 3.2.3).

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.

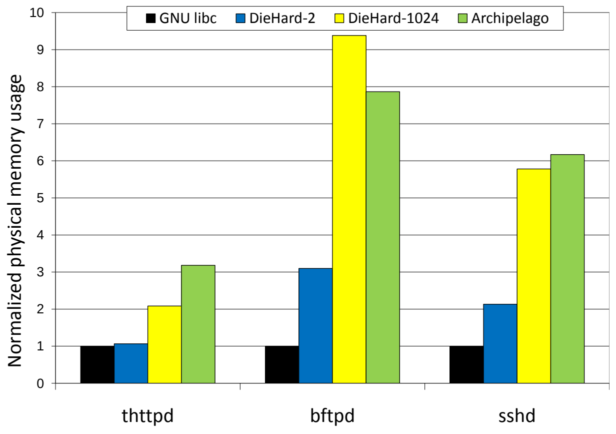
3.2.4 Impact of Coloring

As described in Section 3.1.1, Archipelago's object-per-page allocator uses coloring to reduce cache conflicts. Surprisingly, the impact of coloring is undetectable for 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 objects lies at the beginning of its page, and so each access causes a cache miss because all 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 for all experiments.

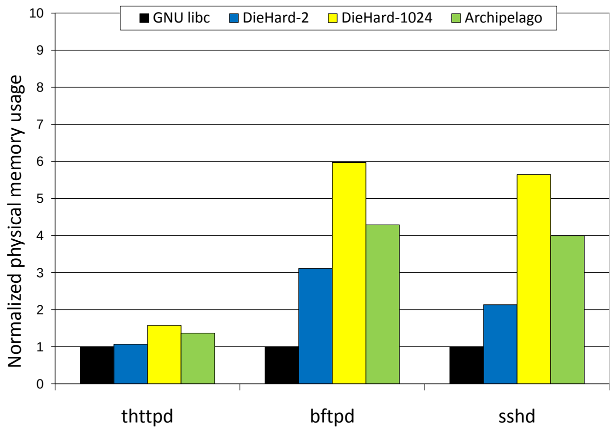
3.2.5 Space Overhead

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 3.9(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 start-up, increasing its virtual memory consumption.



(a) Resident memory usage, without memory pressure.



(b) Resident memory usage, with memory pressure.

Figure 3.9. Resident memory usage with and without memory pressure (Section 3.2.5), 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.

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 3.9(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.

3.2.6 Address Space and Hot Space Sizing

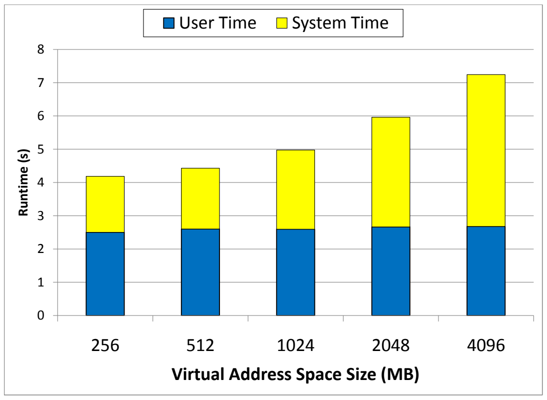
Archipelago's performance is dependent on two user-supplied parameters: the size of the virtual address space used for allocation, as well as the size of the hot object space (i.e., the maximum number of uncompacted pages). Increasing the amount of virtual address space available increases the effectiveness of Archipelago's buffer overflow protection, but at the possible cost of degraded TLB performance and increased page table overhead. Increasing the number 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 3.10 shows how these parameters affect execution time. Varying the hot space size has predictable results: too little 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.

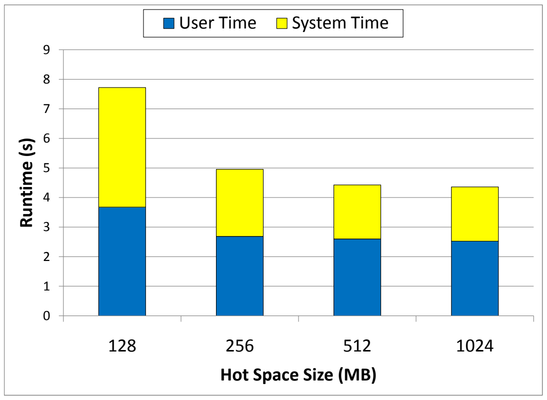
Increasing the amount of virtual address space available to Archipelago shows a consistent trend. A larger virtual address space has little impact on user time, but results in increasing time spent in the kernel. This time is due to a poor fit between the requirements of Archipelago and the current design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

3.2.7 Avoiding Injected Faults

We evaluate the effectiveness of Archipelago in tolerating memory errors by injecting both dangling pointers and buffer overflows. We inject faults into *espresso* running



(a) Varying virtual address space



(b) Varying hot space size

Figure 3.10. Impact of sizing parameters on *espresso* runtime (Section 3.2.6).

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 3.2 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.

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 3.2. The performance of various runtime systems in response to injected memory errors (Section 3.2.7). 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).

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 each system, 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.

3.2.8 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.

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.

3.3 Conclusion

In this chapter, we present Archipelago, a runtime system that provides a high degree of probabilistic memory safety from buffer overflows and dangling pointer errors by spreading objects far apart in virtual address space. It leverages the virtual memory system to control physical memory consumption by compacting infrequently-used objects.

We demonstrate that the overhead of using Archipelago is acceptably low across a range of different server applications, both in terms of CPU performance and memory usage. However, its overhead on applications that allocate many objects on the heap (e.g. a web browser) is high, making it unacceptable for general-purpose use.

CHAPTER 4

DIEHARDER: PROVIDING MAXIMAL PROTECTION WITH MINIMUM OVERHEAD

In this chapter, we present the design and analysis of DieHarder, a memory allocator designed with security as a primary design goal. DieHarder provides a high degree of protection against security vulnerabilities while incurring much lower runtime and memory overhead than Archipelago. As its name implies, DieHarder is based on DieHard, a fully randomized heap allocator designed to improve the resilience of programs to memory errors [9]. Section 1.4.3 discusses DieHard's design and implementation. We first analyze DieHard with respect to security and identify several weaknesses. We then present DieHarder, a secure allocator which combines the best features of Archipelago, DieHard, and OpenBSD. Figure 4.1 presents an overview of DieHarder's architecture.

4.1 DieHard Analysis

While DieHard was designed to increase reliability, it does so by fully randomizing the placement and reuse of heap objects. This randomization makes allocator behavior is highly unpredictable, a primary goal for our secure allocator. In this section, we describe the DieHard allocator and analyze its strengths and weaknesses with respect to our attack models.

Like OpenBSD, DieHard randomizes the placement of allocated objects and the length of time before freed objects are recycled. However, unlike OpenBSD's limited randomization, DieHard randomizes both placement and reuse to the largest practical extent. We show how these two randomization techniques greatly improve protection against attacks by decreasing predictability.

Randomized Placement When choosing where to allocate a new object, DieHard chooses uniformly from every free chunk of the proper size. Furthermore, DieHard's overprovisioning ensures $O(N)$ free chunks, where N is the number of allocated objects. DieHard thus provides $O(\log N)$ bits of entropy for the position of allocated objects, significantly improving on OpenBSD's 4 bits.

This entropy decreases the probability that overflow attacks will succeed. The probability depends upon the limitations of the specific application error. For example, small overflows (at most the size of a single chunk) require that the source object be allocated contiguously with the target chunk.

Theorem 1. *The probability of a small overflow overwriting a specific vulnerable target under DieHard is $O(1/N)$, where N is the number of allocated heap objects* when the later of the source or target chunk was allocated.

Proof. Due to overprovisioning (by a factor of M) there are at least MN free heap chunks to choose for each allocation. Each of these slots is equally likely to be chosen. The probability of the chunks being allocated contiguously is thus at most $2/MN$, assuming free chunks on both sides of the first-allocated chunk (otherwise, the probability is lower). □

The probability of a k-chunk overflow overwriting one of V vulnerable objects generalizes this result. To derive the result, we consider the k object slots following the source object. The first object in V, v_0 has a $(MN - k)/MN$ chance of being outside these k slots, since there are MN possible positions. Each successive v_i has a $(MN - k - i)/MN$ chance, since each $v_0 ... v_{i-1}$ consumes one possible position. Multiplying these probabilities gives

$$\frac{(MN-k)!}{MN \cdot (MN-k-|V|-1)!},$$

the probability of all vulnerable objects residing outside the overwritten region. Thus the overwrite succeeds with probability

$$1 - \frac{(MN-k)!}{MN \cdot (MN-k-|V|-1)!}.$$

If $|V| << N$, each factor is approximately $(MN - k)/MN$, making the probability of a successful attack

$$1 - \left(\frac{(MN-k)}{MN}\right)^{|V|}.$$

Randomized Reuse DieHard chooses the location of newly-allocated chunks randomly across all free chunks of the proper size. Because of its overprovisioning (M-factor), the number of free chunks is always proportional to N, the number of allocated objects. Thus the probability of returning the most-recently-freed chunk is at most $1/MN$. This bound holds even if we continuously allocate without freeing, since the allocator maintains its M overprovisioning factor. In other words, the allocator is sampling with replacement. Thus, like OpenBSD, t follows a geometric distribution with $p = 1/MN$. Unlike OpenBSD, which has low fixed reuse entropy, DieHard provides $O(\log N)$ bits, making reuse much less predictable.

4.2 DieHarder Design and Implementation

As shown in the previous section, DieHard provides greater security guarantees than other general-purpose allocators. However, DieHard was designed for increased reliability against memory errors rather than for increased security. Several features

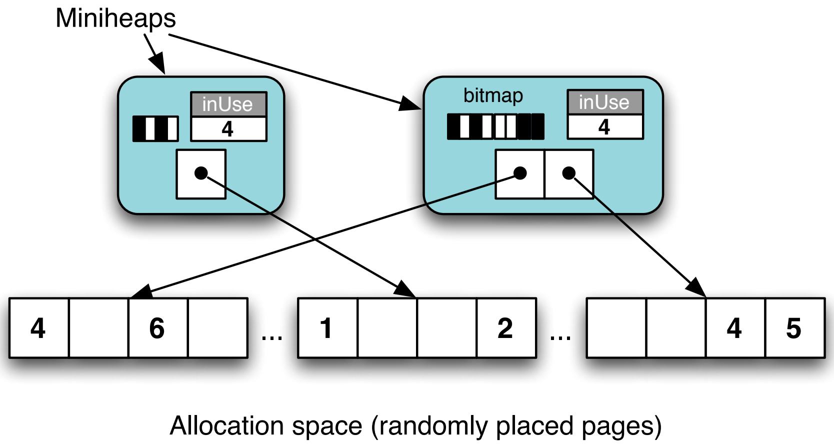


Figure 4.1. An overview of DieHarder's heap layout.

of DieHard enable the program to continue running after experiencing memory errors, rather than thwarting potential attackers. In this section, we describe DieHarder's changes to the original DieHard allocator that substantially enhance its protection against heap-based attacks.

Sparse Page Layout

DieHard's first weakness is its use of large, contiguous regions of memory. Allocating such regions is more efficient than sparse pages, requiring fewer system calls and smaller page tables. This heap layout results in large regions without guard pages, allowing single overflows to overwrite large numbers of heap chunks.

In contrast, OpenBSD's allocator uses a *sparse page layout*, where small objects are allocated within pages spread sparsely across the address space. This approach relies on OpenBSD's ASLR to allocate randomly-placed pages via `mmap`. On 64-bit systems, ASLR makes it highly unlikely that two pages will be adjacent in memory. As a result, a single overflow cannot span a page boundary without hitting unmapped memory and crashing the program.

Our first enhancement to DieHard is to use sparse page allocation. Similarly to OpenBSD, DieHarder randomly allocates individual pages from a large section of address space. DieHarder treats these pages like DieHard, carving them up into size-segregated chunks tracked by an allocation bitmap. Allocation is also performed as in DieHard, with an extra level of indirection to cope with sparse page mapping.

Object deallocation is more complicated, since finding the correct bitmap given an object address is not straightforward. DieHard finds the correct metadata using a straightforward search, exploiting its heap layout to require expected constant time. With sparse pages, however, using DieHard's approach would require $O(N)$ time. DieHarder instead uses a hash table to store references to page metadata, ensuring constant-time `free` operations.

Address Space Sizing

To achieve full randomization under operating systems that randomize only the base address of the `mmap` region, such as Linux, DieHarder explicitly randomizes the addresses of small object pages. It does so by mapping a large, fixed-size region of virtual address space and then sparsely using individual pages. This implementation wastes a large amount of virtual memory, but uses physical memory efficiently, since most virtual pages are not backed by physical page frames.

While the size of the virtual region does not affect the amount of physical memory used by application data, it does affect the size of the process's page tables. The x86-64 uses a 4-level page table. Contiguous allocations of up to 1 GB (2^{18} pages) require only 1 or 2 entries in the top three levels of the table, consuming approximately 512 pages or 2 MB of memory for the page table itself. In contrast, sparsely allocating 1 GB of pages within the full 48-bit address space requires mostly-complete middle levels of the table. Each 512-entry second-level page-middle directory (PMD) spans 1 GB, and the expected number of pages contained within each 1 GB region is 1. The resulting page table would thus require on the order of $2 \cdot 2^{18}$ page table entries (PTEs) and PMDs, for a staggering 2 GB page table.

Even if physical memory is not an issue, these sparse page tables can drastically decrease cache efficiency when the application's working set exceeds the TLB reach. When each PMD and PTE is sparse, the cache lines containing the actual entries have only 1/8 utilization (8 of 64 bytes). Combined with needing a line for each PMD and PTE, the effective cache footprint for page tables grows by 16× under a sparse layout.

To combat this effect, we restrict DieHarder's randomization to a smaller virtual address range.

Destroy-on-free

DieHarder, like many debugging allocators, fills freed objects with random data. While this policy empirically helps find memory errors, within the context of DieHarder, it is required to limit the effectiveness of certain attack strategies.

Unlike allocators with deterministic reuse, repeated `malloc` and `free` operations in DieHarder return different chunks of memory. If freed objects were left intact, even an attacker with limited control of heap operations (e.g., only able to hold only one object live at a time) could fill an arbitrary fraction of the heap with attacker-controlled data by exploiting random placement. In the same scenario, overwriting the contents of freed objects ensures only one chunk at a time contains attacker-controlled data.

4.3 DieHarder Analysis

Using a sparse page heap layout provides greater protection against heap overflow attacks and heap spraying. Unlike DieHard, DieHarder does not allocate small objects on contiguous pages.

Overflows

The sparse layout provides two major protections against overflow attacks. First, because pages are randomly distributed across a large address space, the probability of allocating two contiguous pages is low. Thus, pages are protected by guard pages on both sides with high probability. Overflows past the end of a page will hit the guard page, causing the attack to fail.

The chance of hitting a guard page depends on H, the number of allocated pages and S, the size in pages of DieHarder's allocated virtual address space. The chance of having a guard page after any allocated page is $(S - H)/S$. This probability increases with S; however, large values of S can degrade performance, as discussed in Section 4.2.

Combined with randomized object placement, the memory immediately after *every* allocated object has a significant probability of being unmapped. The worst case for DieHarder is 16-byte objects, since there are 256 16-byte chunks per page. The probability of a 1-byte overflow crashing immediately is at least

$$\frac{(S-H)}{S} \cdot \frac{1}{256}.$$

The first term represents the probability of the following page being unmapped, and the second term the probability of the overflowed object residing in the last slot on the page.

Heap Spraying

DieHarder's sparse layout protects against heap spraying attacks by providing more entropy in object addresses. DieHarder's fully-randomized allocation eliminates dependence between the addresses of objects on different pages. The number of objects that are easily guessable given a valid object address is limited to the number

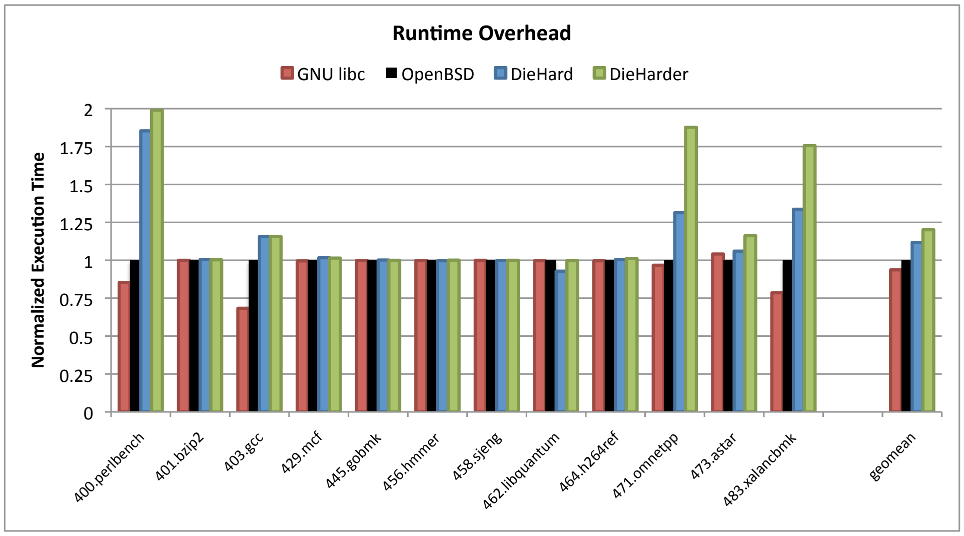


Figure 4.2. Runtime overhead of the different allocators, normalized to their runtime using OpenBSD's allocator. In exchange for a substantial increase in entropy, DieHarder imposes on average a 20% performance penalty vs. OpenBSD for CPU-intensive benchmarks, though it has no performance impact on Firefox (see Section 4.4).

that reside on a single page, which is further reduced by DieHarder's overprovisioning factor (inherited from DieHard).

4.4 DieHarder Evaluation

We measure the runtime overhead of DieHarder compared to four existing allocators, GNU libc (based on DLmalloc 2.7), DLmalloc 2.8.4, DieHard, and OpenBSD. We enabled DLmalloc 2.8's object footers that improve its resilience against invalid frees. We use the adaptive version of DieHard [10] (version 1.1). To isolate allocator effects, we ported OpenBSD's allocator to Linux. We run DieHarder using a 4 GB virtual address space for randomizing small object pages. We discuss the impact of this design parameter in Section 4.2.

Our experimental machine is a single-socket, quad-core Intel Xeon E5520 (Nehalem) running at 2.27GHz with 4 GB of physical memory. We first evaluate the CPU overhead of various allocators using the SPECint2006 benchmark suite. Unlike its predecessor (SPECint2000), this suite places more stress on the allocator, containing a number of benchmarks with high allocation rates and large heap memory footprints.

Figure 4.2 shows the runtime of the benchmark suite using each allocator, normalized to its runtime under OpenBSD's allocator. DieHarder's overhead varies from -7% to 117%, with a geometric mean performance impact of 20%. Most benchmarks exhibit very little overhead (less than 2%). The benchmarks that suffer the most, `perlbench`, `omnetpp`, and `xalancbmk`, significantly stress the allocator due to their unusually high allocation rates.

Firefox In addition to the SPECint2006 suite, we evaluated the performance of the Firefox browser using both DieHarder (4 GB virtual address space) and GNU libc. In order to precisely measure Firefox's performance, we used the Selenium browser automation tool to automatically load a sequence of 20 different web pages. We used an offline proxy, wwwoffle, to minimize the effect of network latency and ensure identical behavior across all experiments. We repeated this experiment 15 times for each allocator.

The results show no statistically significant difference in performance between allocators at the 5% level. The mean runtimes for GNU libc and DieHarder, respectively, were 44.2 and 41.6 seconds, with standard deviations of 7.13 and 6.12. This result qualitatively confirms that DieHarder is practical for use.

CHAPTER 5

EXTERMINATOR: PROBABILISTICALLY ISOLATING BUFFER OVERFLOW AND DANGLING POINTER ERRORS

DieHarder significantly reduces the probability of success of each individual attack attempt. Given enough time, however, an attacker will eventually succeed. Waiting for the vendor to patch the vulnerability may require too much time. According to Symantec, the average time between the discovery of a critical, *remotely exploitable* memory error and the release of a patch for enterprise applications is 28 days [78].

Users are thus faced with a difficult tradeoff: sacrifice availability and stop using the application until a fix is available, or sacrifice security and take the risk of being exploited. A business relying on vulnerable software for e-commerce, for example, clearly cannot shut down for a month. Other applications, such as Internet Explorer or Adobe Flash Player, are used by millions of people worldwide, meaning that zero-day exploits with low probability of success will compromise thousands of machines before the vendor is even aware of the vulnerability.

To cope with these situations, we present Exterminator, a runtime system that automatically isolates and corrects buffer overflows and dangling pointer errors. Exterminator requires neither source code nor programmer intervention, and fixes existing errors without introducing new ones. By automatically correcting errors, Exterminator increases reliability and mitigates the tradeoff between availability and security. To our knowledge, this system is the first of its kind.

Exterminator relies on an efficient probabilistic debugging allocator that we call *DieFast*. DieFast is based on DieHard's allocator [9], which ensures that heaps are independently randomized. However, while DieHard can only probabilistically tolerate errors, DieFast probabilistically detects them [1].

Exterminator can operate in three distinct modes: an *iterative mode* for repeated runs over the same input, a *replicated mode* that can correct errors on the fly by running multiple redundant executions, and a *cumulative mode* that corrects errors across multiple runs of the same application with different inputs.

When Exterminator discovers an error, it produces a *heap image* that contains the complete state of the heap, similar to a core file. Exterminator's *probabilistic error isolation* algorithms then process one or more heap images (depending on the execution mode) to locate the source and size of buffer overflows and dangling pointer errors. These algorithms rely on sound statistical inference methods, not *ad hoc* heuristics, and have provably low false positive and false negative rates.

Once Exterminator locates a buffer overflow, it determines the allocation site of the overflowed object (based on the context-sensitive callsite to `malloc`) using heap metadata, and the size of the overflow by the amount of detected heap corruption. For dangling pointer errors, Exterminator determines both the allocation and deletion sites of the dangled object, and estimates how prematurely the object was freed.

With this information in hand, Exterminator corrects the errors by generating *runtime patches*. These patches operate in the context of a *correcting allocator*. The correcting allocator prevents overflows by padding objects and prevents dangling pointer errors by deferring object deallocations based on the data in the runtime patch. These actions impose little runtime or space overhead because Exterminator's runtime patches are specific to a single allocation site for each buffer overflow, and a single allocation/deallocation site pair for each dangling pointer.

[1] DieFast's implementation predates DieHarder and is thus based upon DieHard. DieFast could alternatively use DieHarder's heap layout, which would improve the convergence rates for Exterminator's cumulative mode isolation algorithms.

After Exterminator completes patch generation, it creates or updates a patch to correct the bug in subsequent executions. If running in replicated mode, it triggers a patch update in the running processes to fix the bug in the current execution. Exterminator's patches compose straightforwardly, enabling *collaborative bug correction*: users running Exterminator can automatically merge their patches, thus systematically and continuously improving application reliability.

While Exterminator's allocation padding can prevent minor programming errors, such as off-by-ones, it cannot deterministically prevent security vulnerabilities resulting from arbitrary, unbounded overflows. However, for many vulnerabilities, Exterminator can detect, isolate, and adapt the application well before the attack succeeds, with high probability. This adaptation forces attackers to cope with moving targets, greatly increasing the difficulty of mounting a successful attack.

5.1 Software Architecture

Exterminator's software architecture extends and modifies DieHard to enable its error isolating and correcting properties. Section 1.4.3 described DieHard and its allocator. This section first describes how Exterminator augments DieHard's heap layout to track information needed to identify and remedy memory errors. Second, it presents DieFast, a probabilistic debugging allocator that exposes errors to Exterminator. Finally, it describes Exterminator's three modes of operation.

5.1.1 Exterminator's Heap Layout

Figure 5.1 illustrates Exterminator's heap layout, which includes five fields per object for error isolation and correction: an **object id**, **allocation** and **deallocation sites**, **deallocation time**, which records when the object was freed, and a **canary bitset** that indicates if the object was filled with canaries (Section 5.1.2).

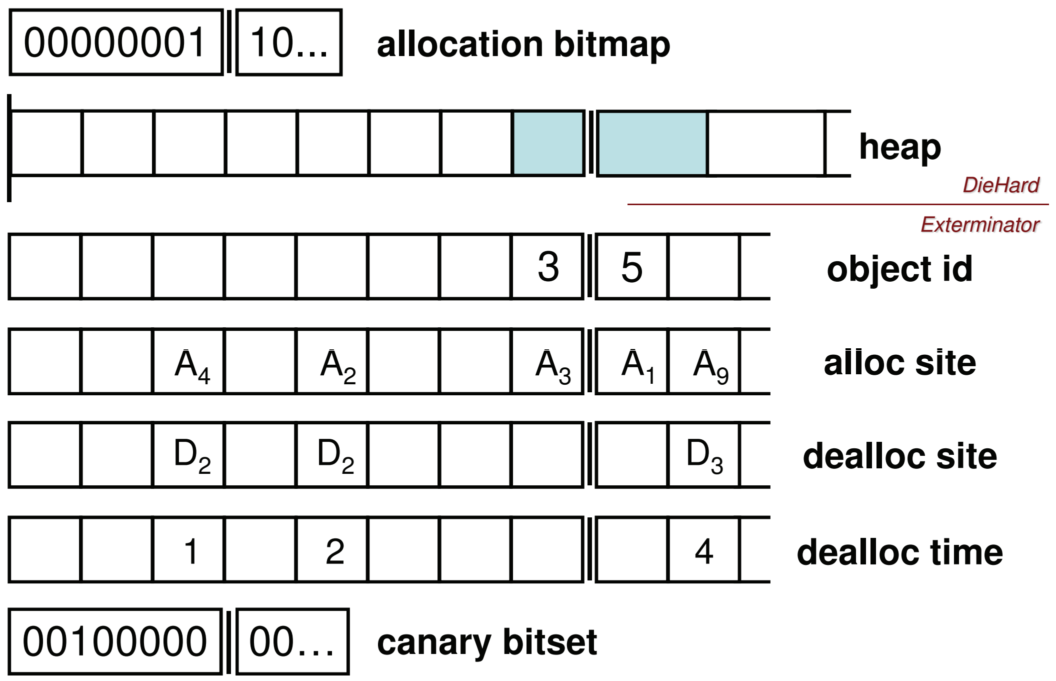


Figure 5.1. An abstract view of Exterminator's heap layout. Metadata below the horizontal line contains information used for error isolation and correction (see Section 5.1.1).

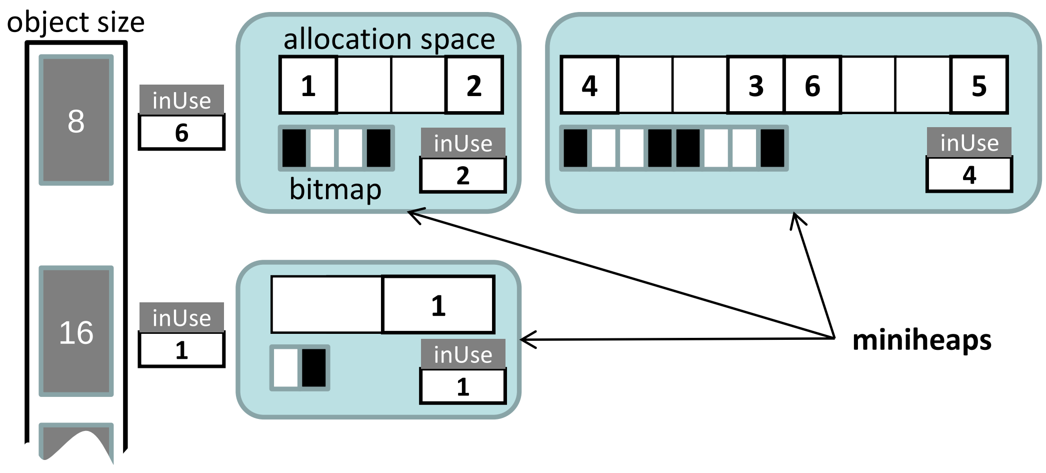


Figure 5.2. The adaptive DieHard heap layout, used by Exterminator. Objects in the same size class are allocated randomly from separate *miniheaps*, which combined hold M times more memory than required (here, $M = 2$).

An object id of n means that the object is the nth object allocated. Exterminator uses object ids to identify objects across multiple heap images. These ids are

```
int computeHash (int * pc)
  int hash = 5381;
  for (int i = 0; i < 5; i++)
    hash = ((hash << 5) + hash) + pc[i];
  return hash;
```

Figure 5.3. Site information hash function, used to store allocation and deallocation call sites (see Section 5.1.1).

needed because the object's address cannot be used to identify it across differently-randomized heaps.

The site information fields capture the calling context for allocations and deallocations. For each, Exterminator hashes the least significant bytes of the five most-recent return addresses into 32 bits using the DJB2 hash function [12] (see Figure 5.3).

This out-of-band metadata accounts for 16 bytes plus two bits (one each for the allocation and canary bitmaps) of space overhead for every object. This overhead is comparable to that of typical freelist-based memory managers like the Lea allocator, which prepend 8-byte (on 32-bit systems) or 16-byte headers (on 64-bit systems) to allocated objects [44]. Exterminator's metadata does not contain raw pointer values and can use 32-bit words even on 64-bit systems.

5.1.2 DieFast: A Probabilistic Debugging Allocator

Exterminator uses a new, probabilistic debugging allocator that we call DieFast. DieFast uses the same randomized heap layout as DieHard, but extends its allocation and deallocation algorithms to detect and expose errors. Figure 5.4 presents pseudo-code for the DieFast allocator. Unlike previous debugging allocators, DieFast has a number of unusual characteristics tailored for its use in the context of Exterminator.

```
void * diefast_malloc (size_t sz) {
  void * ptr = really_malloc (sz);
  // Check if the object wasn't
  // canary-filled or is uncorrupted.
  bool ok = verifyCanary (ptr);
  if (!ok) { mark allocated; signal error }
  return ptr;
}
```

```
void diefast_free (void * ptr) {
  really_free (ptr);
  // Check preceding and following objects.
  bool ok = true;
  if (isFree (previous (ptr)))
    ok &= verifyCanary (previous(ptr));
  if (isFree (next(ptr)))
    ok &= verifyCanary (next(ptr));
  if (!ok) { signal error; }
  // Probabilistically fill with canary.
  if (notCumulativeMode || random() < p)
    fillWithCanary (ptr);
}
```

Figure 5.4. Pseudo-code for DieFast, a probabilistic debugging allocator (Section 5.1.2).

Implicit Fence-posts

Many existing debugging allocators pad allocated objects with fence-posts (filled with **canary** values) on both sides. They can thus detect buffer overflows by checking the integrity of these fence-posts. This approach has the disadvantage of increasing space requirements. Combined with the already-increased space requirements of a DieHard-based heap, the additional space overhead of padding may be unacceptably large.

DieFast exploits two facts to obtain the effect of fence-posts without any additional space overhead. First, because its heap layout is headerless, one fence-post serves double duty: a fence-post following an object can act as the one preceding the next object. Second, because allocated objects are separated by $E(M-1)$ freed objects on the heap, we use freed space to act as fence-posts.

Random Canaries

Traditional debugging canaries include values that are readily distinguished from normal program data in a debugging session, such as the hexadecimal value `0xDEADBEEF`. However, one drawback of a deterministically-chosen canary is that it is always possible for the program to use the canary pattern as a data value. Because DieFast uses canaries located in freed space rather than in allocated space, a fixed canary would lead to a high false positive rate if that data value were common in allocated objects.

DieFast instead uses a random 32-bit value set at startup. Since both the canary and heap addresses are random and differ on every execution, any fixed data value has a low probability of colliding with the canary, thus ensuring a low false positive rate (see Theorem 3). To increase the likelihood of detecting an error, DieFast sets the last bit of the canary. Setting this bit will cause an alignment error if the canary is dereferenced, but keeps the probability of an accidental collision with the canary low ($1/2^{31}$).

Probabilistic Fence-posts

Intuitively, the most effective way to expose a dangling pointer error is to fill all freed memory with canary values. For example, dereferencing a canary-filled pointer will likely trigger a segmentation violation.

Unfortunately, reading random values does not necessarily cause programs to fail. For example, in the `espresso` benchmark, some objects hold bitsets. Filling a freed bitset with a random value does not cause the program to terminate but only affects the correctness of the computation.

If reading from a canary-filled dangling pointer causes a program to diverge, there is no way to narrow down the error. In the worst-case, half of the heap could be filled with freed objects, all overwritten with canaries. All of these objects would then be potential sources of dangling pointer errors.

In cumulative mode, Exterminator prevents this scenario by non-deterministically writing canaries into freed memory randomly with probability p, and setting the appropriate bit in the canary bitmap. This probabilistic approach may seem to degrade Exterminator's ability to find errors. However, it is required to isolate read-only dangling pointer errors, where the canary itself remains intact. Because it would otherwise take an impractically large number of iterations or replicas to isolate these errors, Exterminator always fills freed objects with canaries when not running in cumulative mode (see Sections 5.3.2 and 5.5.2.1 for discussion).

Probabilistic Error Detection

Whenever DieFast allocates memory, it examines the memory to be returned to verify that any canaries are intact. If not, in addition to signaling an error (see Section 5.1.3), DieFast sets the allocated bit for this chunk of memory. This "bad object isolation" ensures that the object will not be reused for future allocations, preserving its contents for Exterminator's subsequent use. Checking canary integrity on each allocation ensures that DieFast will detect heap corruption within $E(H)$ allocations, where H is the number of objects on the heap.

After every deallocation, DieFast checks both the preceding and subsequent objects. For each of these, DieFast checks if they are free. If so, it performs the same canary check as above. Recall that because DieFast's allocation is random, the identity of these adjacent objects will differ from run to run. Checking the predecessor and successor on each free allows DieFast to detect buffer overruns immediately upon object deallocation.

5.1.3 Modes of Operation

Exterminator can be used in three modes of operation: an iterative mode suitable for testing or whenever all inputs are available, a replicated mode that is suitable both for testing and for restricted deployment scenarios, and a cumulative mode that is suitable for broad deployment. All of these rely on the generation of heap images, which Exterminator examines to isolate errors and compute runtime patches.

If Exterminator discovers an error when executing a program, or if DieFast signals an error, Exterminator forces the process to emit a heap image file. This file is akin to a core dump, but contains less data (e.g., no code), and is organized to simplify processing. In addition to the full heap contents and heap metadata, the heap image includes the current allocation time (measured by the number of allocations to date).

Iterative Mode

Exterminator's iterative mode operates without replication. To find a single bug, Exterminator is initially invoked via a command-line option that directs it to stop as soon as it detects an error. Exterminator then re-executes the program in "replay" mode over the same input (but with a new random seed). In this mode, Exterminator reads the allocation time from the initial heap image to abort execution at that point; we call this a **malloc breakpoint**. Exterminator then begins execution and ignores DieFast error signals that are raised before the malloc breakpoint is reached.

Once it reaches the malloc breakpoint, Exterminator triggers another heap image dump. This process can be repeated multiple times to generate independent heap images. Exterminator then performs post-mortem error isolation and runtime patch generation. A small number of iterations usually suffices for Exterminator to generate runtime patches for an individual error, as we show in Section 5.5.2. When run with a correcting memory allocator that incorporates these changes (described in detail in Section 5.4.3), these patches automatically fix the isolated errors.

This mode identifies objects by their allocation time, which is effectively a serial number. These numbers must be consistent across multiple runs of the program, which means that execution must be deterministic. Iterative mode thus cannot be used on unaltered multithreaded programs, since they may allocate objects with a different global order during different runs. Combining DieFast with a runtime that guarantees deterministic execution, such as CoreDet [6] or Grace [8], would allow Exterminator's iterative mode to function with multithreaded programs.

Replicated Mode

The iterated mode described above works well when all inputs are available so that re-running an execution is feasible. However, when applications are deployed in the field, such inputs may not be available, and replaying may be impractical. The replicated mode of operation allows Exterminator to correct errors while the program is running, without the need for multiple iterations.

Like DieHard, Exterminator can run a number of differently-randomized replicas simultaneously (as separate processes), broadcasting inputs to all and voting on their outputs. However, Exterminator uses DieFast-based heaps, each with a correcting allocator. This organization lets Exterminator discover and fix errors.

In replicated mode, when DieFast signals an error or the voter detects divergent output, Exterminator sends a signal that triggers a heap image dump for each replica. If the program crashes because of a segmentation violation, a signal handler also dumps a heap image.

If DieFast signals an error, the replicas that dump a heap image do not have to stop executing. If their output continues to be in agreement, they can continue executing concurrently with the error isolation process. When the runtime patch generation process is complete, that process signals the running replicas to tell the

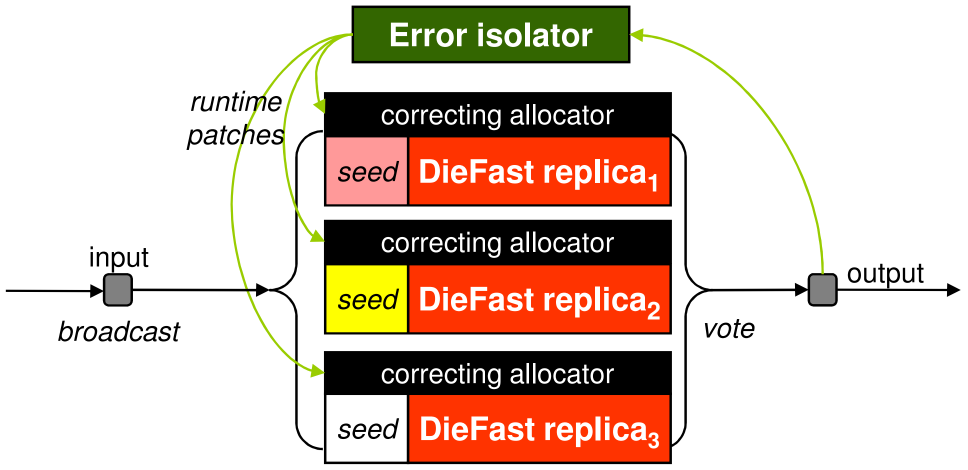


Figure 5.5. Exterminator's replicated architecture (Section 5.1.3). Replicas are equipped with different seeds that fully randomize their DieFast-based heaps (Section 5.1.2), input is broadcast to all replicas, and output goes to a voter. A crash, output divergence, or signal from DieFast triggers the error isolator (Section 5.2), which generates *runtime patches*. These patches are fed to correcting allocators (Section 5.4), which fix the bug for current and subsequent executions.

correcting allocators to reload their runtime patches. Thus, subsequent allocations in the same process will be patched on-the-fly without interrupting execution.

Exterminator's replicated mode has the same determinism requirements as iterative mode, and thus the same implications for use with multithreaded programs.

Cumulative Mode

While the replicated mode can isolate and correct errors on-the-fly in deployed applications, it may not be practical in all situations. For example, replicating applications with high resource requirements may cause unacceptable overhead. In addition, multi-threaded or non-deterministic applications can exhibit different allocation activity and so cause object ids to diverge across replicas. To support these applications, Exterminator uses its third mode of operation, **cumulative** mode, which isolates errors without replication or multiple identical executions.

When operating in cumulative mode, Exterminator reasons about objects grouped by allocation and deallocation sites instead of individual objects, since objects are no longer guaranteed to be identical across different executions.

Because objects from a given site only occasionally cause errors, often at low frequencies, Exterminator requires more executions than in replicated or iterative mode in order to identify these low-frequency errors without a high false positive rate. Instead of storing heap images from multiple runs, Exterminator computes relevant statistics about each run and stores them in its patch file. The retained data is on the order of a few kilobytes per execution, compared to tens or hundreds of megabytes for each heap image.

5.2 Iterative and Replicated Error Isolation

Exterminator employs two different families of error isolation algorithms: one set for replicated and iterative modes, and another for cumulative mode.

When operating in its replicated or iterative modes, Exterminator's probabilistic error isolation algorithm operates by searching for discrepancies across multiple heap images. Exterminator relies on corrupted canaries to indicate the presence of an error. A corrupted canary (one that has been overwritten) can mean two things: if every object has the same corruption, then it is likely a dangling pointer error, as Theorem 2 shows. If canaries are corrupted in multiple objects, then it is likely to be a buffer overflow. Exterminator limits the number of false positives for both overflows and dangling pointer errors.

5.2.1 Buffer Overflow Detection

Exterminator examines heap images looking for discrepancies across the heaps, both in overwritten canaries and in live objects. If an object is not equivalent across the heaps (see below), Exterminator considers it to be a candidate *victim* of an overflow.

To identify victim objects, Exterminator compares the contents of both objects identified by their object id across all heaps, word-by-word. Exterminator builds an *overflow mask* that comprises the discrepancies found across all heaps. However, because the same logical object may legitimately differ across multiple heaps, Exterminator must take care not to consider these as overflows.

First, a freed object may differ across heaps because it was filled with canaries only in some of the heaps. Exterminator uses the canary bitmap to identify this case.

Second, an object can contain pointers to other objects, which are randomly located on their respective heaps. Exterminator uses both deterministic and probabilistic techniques to distinguish integers from pointers. Briefly, if a value interpreted as a pointer points inside the heap area and points to the same logical object across all heaps, then Exterminator considers it to be the same logical pointer, and thus not a discrepancy. Exterminator also handles the case where pointers point into dynamic libraries, which newer versions of Linux place at random base addresses.

Finally, an object can contain values that legitimately differ from process to process. Examples of these values include process ids, file handles, and pseudorandom numbers. Some data structures may have different topologies depending on random allocation choices, such as a red-black tree using pointers as keys. When Exterminator finds an object field that differs in an unexplainable way across all heaps, it assumes that the difference is legitimate and not caused by heap corruption. This heuristic is required to eliminate false positives, but may cause false negatives if the object is corrupted in *all* heaps.

For small overflows, the risk of missing an overflow by ignoring overwrites of the same objects across multiple heaps is low:

Theorem 2. *Let k be the number of heap images, S the length (*in number of objects*) of the* overflow string*, and H the number of objects on the heap. Then the probability of an overflow overwriting an object on all k heaps is:*

$$P(\textit{identical overflow}) \;\leq\; H \times (S/H)^k.$$

Proof. This result holds for a stronger adversary than usual—rather than assuming a single contiguous overflow, we allow an attacker to arbitrarily overwrite any S distinct objects. Consider a given object a. On each heap, S objects are corrupted at random. The probability that object i is corrupted on a single heap is (S/H). Call E_i the event that object i is corrupted across all heaps; the probability $P(E_i)$ is $(S/H)^k$. The probability that at least one object is corrupted across all the heaps is $P(\cup_i E_i)$, which by a straightforward union bound is at most $\sum_i P(E_i) = H \times (S/H)^k$. □

We now bound the worst-case false negative rate for buffer overflows; that is, the odds of not finding a buffer overflow because it failed to overwrite any canaries.

Theorem 3. *Let M be the heap multiplier, so a heap is never more than $1/M$ full. The likelihood that an overflow of length b bytes fails to be detected by comparison against a canary is at most:*

$$P(\textit{missed overflow}) \;\leq\; \left(1 - \frac{M-1}{2M}\right)^k + \frac{1}{256^b}.$$

Proof. Each heap is at least $(M-1)/M$ free. Since DieFast fills free space with canaries with $P = 1/2$, the fraction of each heap filled with canaries is at least $(M-1)/2M$. The likelihood of a random write not landing on a canary across all k heaps is thus at most $(1-(M-1)/2M)^k$. The overflow string could also match the canary value. Since the canary is randomly chosen, the odds of this are at most $(1/256)^b$. □

Culprit Identification

At this point, Exterminator has identified the possible victims of overflows. For each victim, it scans the heap images for a matching **culprit**, the source of the overflow into a victim. Because Exterminator assumes that overflows are deterministic when operating in iterative or replicated modes, the culprit must be the same distance δ bytes away from the victim in every heap image. In addition, Exterminator requires that the overflowed values have some bytes in common across the images, and ranks them by their similarity.

Exterminator checks every other heap image for the candidate culprit, and examines the object that is the same δ bytes forwards. If that object is free and should be filled with canaries but they are not intact, then it adds this culprit-victim pair to the candidate list.

We now bound the false positive rate. Because buffer overflows can be discontiguous, every object in the heap that precedes an overflow is a potential culprit. However, each additional heap dramatically lowers this number:

Theorem 4. *The expected number of objects (possible culprits) the same distance δ from any given victim object across k heaps is:*

$$\textit{E(possible culprits)} \;=\; \frac{1}{(H-1)^{k-2}}.$$

Proof. Without loss of generality, assume that the victim object occupies the last slot in every heap. An object can thus be in any of the remaining $n = H - 1$ slots. The odds of it being in the same slot in k heaps is $p = 1/(H-1)^{k-1}$. This is a binomial distribution, so E(possible culprits) $= np = 1/(H-1)^{k-2}$. $\square$

With only one heap image, all $(H-1)$ objects are potential culprits, but one additional image reduces the expected number of culprits for any victim to just 1 $(1/(H-1)^0)$, effectively eliminating the risk of false positives.

Once Exterminator identifies a culprit-victim pair, it records the overflow size for that culprit as the maximum of any observed δ to a victim. Exterminator also assigns each culprit-victim pair a score that corresponds to its confidence that it is an actual overflow. This score is $1-(1/256)^S$, where S is the sum of the length of detected overflow strings across all pairs. Intuitively, small overflow strings (e.g., one byte) detected in only a few heap images are given lower scores, and large overflow strings present in many heap images get higher scores.

After overflow processing completes and at least one culprit has a non-zero score, Exterminator generates a runtime patch for an overflow from the most highly-ranked culprit.

5.2.2 Dangling Pointer Isolation

Isolating dangling pointer errors falls into two cases: a program may *read and write* to the dangled object, leaving it partially or completely overwritten, or it may only *read* through the dangling pointer. Exterminator does not handle read-only dangling pointer errors in iterative or replicated mode because it would require too many replicas (e.g., around 20; see Section 5.5.2.1). However, it handles overwritten dangling objects straightforwardly.

When a freed object is overwritten with identical values across multiple heap images, Exterminator classifies the error as a dangling pointer overwrite. As Theorem 2 shows, this situation is highly unlikely to occur for a buffer overflow. Exterminator then generates an appropriate runtime patch, as Section 5.4.2 describes.

5.3 Cumulative Error Isolation

When operating in cumulative mode, Exterminator isolates memory errors by computing summary information accumulated over multiple executions, rather than by operating over multiple heap images. This mode lets Exterminator isolate memory errors without the need for replication, identical inputs, or deterministic execution.

5.3.1 Buffer Overflow Detection

Exterminator's cumulative mode buffer overflow isolation algorithm proceeds in three phases. First, it identifies heap corruption by looking for overwritten canary values. Second, for each allocation site, it computes an estimate of the probability that an object from that site could be the source of the corruption. Third, it combines these independent estimates from multiple runs to identify sites that consistently appear as candidates for causing the corruption.

After computing the set of corrupt object slots, Exterminator examines allocation sites and finds possible culprits. To reason about an individual allocation site, Exterminator must consider all objects allocated from that site. We use Bayesian inference to compare two hypotheses, H_0, the allocation site does not produce overflowed objects, and H_1, the allocation site produces some overflowed objects.

An object that causes corruption by a forward overflow (i.e., it corrupts memory at a higher address) must satisfy two criteria. First, it must lie on the same miniheap as the corruption. Because miniheaps are randomly located throughout the whole address space, we assume that the probability that an overflow crosses miniheap boundaries to cause corruption without first causing a segmentation violation is negligible. Second, the overflowed object must lie at a lower address than the corruption. Backwards overflows (underruns) are handled independently using the same algorithm with the obvious adaptations.

For each object, the error isolation algorithm computes the conditional probability that the object satisfies the criteria given the null hypothesis (i.e., that the object is not the source of the corruption). The total probability is the product of the probabilities of being allocated in the same miniheap (the left-hand term below), times the probability of it falling on the left side of the corruption (the right-hand term). The first term is the size of the corrupt miniheap, divided by the sum of the sizes of all miniheaps available in the size class at the time the object was allocated (The size' function below ignores miniheaps that did not exist at the time of the object's allocation). Let M_c be the corrupted miniheap, k the index of the corrupted slot in M_c, $\tau(i)$ and $\tau(M_j)$ the allocation time of object i or miniheap M_j, respectively, and $size(M_i)$ the number of object slots in miniheap M_i. The probability $P(C_i)$ that object i satisfies the criteria is then:

$$P(C_i) = \frac{size'(i, M_c)}{\sum_{M_j} size'(i, M_j)} \cdot \frac{k}{size(M_c)}$$

where

$$size'(i, M_j) = \begin{cases} 0 & \tau(M_j) > \tau(i) \\ size(M_j) & \tau(M_j) \leq \tau(i). \end{cases}$$

For each allocation site A, Exterminator then computes the probability $P(C_A)$ that at least one object from the site satisfied the criteria (1 minus the probability of all objects *not* satisfying) as

$$P(C_A) = 1 - \left(\prod_{i \text{ from } A} \big(1 - P(C_i)\big) \right).$$

This value $P(C_A)$, combined with the actual observed value C_A, is the complete summary that Exterminator computes and stores between runs. Intuitively, each run can be thought of as a coin flip, where $P(C_A)$ is the probability of heads, and $C_A = 1$ if the coin flip resulted in heads.

Using the estimates from multiple runs, Exterminator then identifies allocation sites that satisfy the criteria more often than expected under the null hypothesis. These allocation sites are those that generate overflowed objects. Let θ_A be the probability that an observed corrupted object was caused by an overflow from an object allocated from site A. For sites with no overflow errors, $\theta_A = 0$. For sites with errors, θ_A is some value greater than zero, depending on the number of other bugs in the program. The algorithm compares the likelihoods of the two competing hypotheses: $H_0 : \theta_A = 0$ (no overflowed objects), and $H_1 : \theta_A > 0$ (some overflowed objects).

Exterminator's error classifier takes as input the sequence of computed probabilities $X_i = P(C_A)$ and the observed values $Y_i = C_A$ from each run. Using a Bayesian model, Exterminator rejects H_0 and identifies A as an error source when $P(H_1|\bar{X}, \bar{Y}) > P(H_0|\bar{X}, \bar{Y})$. This condition is equivalent (using Bayes' rule) to

$$\frac{P(\bar{X}, \bar{Y}|H_1)}{P(\bar{X}, \bar{Y}|H_0)} > \frac{P(H_0)}{P(H_1)}.$$

Because the true prior probabilities of the hypotheses are unknown, Exterminator estimates them. Different estimates trade off between false positive rate and the number of runs required to identify true errors. Using a ratio of prior probabilities $P(H_0)/P(H_1) = 1/cN$, where N is the total number of allocation sites and c a small constant (currently, $c = 4$) generally produces a well-behaved, conservative classifier. This prior is reasonable because there is some probability that the corruption was caused by an overflow (as opposed to a dangling pointer), represented by the $1/c$ factor, and a small probability that each allocation site is the culprit (the $1/N$ factor).

Finally, Exterminator computes the above values and compares them. Assuming H_0, each independent run i has a $X_i = P(C_A)$ chance that $Y_i = 1$. By the product rule,

$$P(\bar{X},\bar{Y}|H_0) = \prod_i \big((1-X_i)(1-Y_i) + X_iY_i\big).$$

Computing the likelihood of H_1 requires consideration of all possible values of θ_A. The probability of Y_i is then the causation probability θ_A, plus the probability due to random chance, $(1-\theta_A)X_i$. We assume a uniform prior distribution on θ_A, that is,

$$P(\theta_A) = \begin{cases} 1 & 0 < \theta_A \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

The likelihood is then:

$$P(\bar{X},\bar{Y}|H_1) = \int_0^1 \prod_i \left(\begin{array}{c} \big(1-(1-\theta_A)\,X_i - \theta_A\big)\big(1-Y_i\big) \\ +\,\big((1-\theta_A)X_i + \theta_A\big)Y_i \end{array} \right) d\theta_A.$$

Once Exterminator identifies an erroneous allocation site A, it produces a runtime patch that corrects the error. To find the correct padding value, it searches backwards from the corruption found during the current run until it finds an object allocated from A. It then uses the distance between that object and the end of the corruption as the padding value.

5.3.2 Dangling Pointer Isolation

As with buffer overflows, dangling pointer isolation proceeds by computing summary information over a number of runs. To force each run to have a different effect, Exterminator fills freed objects with canaries with some probability p, turning every execution into a series of Bernoulli trials. If overwriting a prematurely-freed object with canaries leads to an error, then its overwrite will correlate with a failed execution with probability greater than p. Conversely, if an object was not prematurely freed, then overwriting it with canaries should have no correlation with the failure or success of the program.

For each failed run, Exterminator computes the probability that an object was canaried from each allocation site. As in the buffer overflow case, the summary information required is simply this probability (X_i) and whether or not a canary was observed (Y_i).

Because the meaning of this data is the same as in the buffer overflow algorithm, Exterminator uses the same hypothesis test to compute the likelihood that each allocation site is the source of a dangling pointer error.

The choice of p reflects a tradeoff between the precision of the buffer overflow algorithm and dangling pointer isolation. Since overflow isolation relies on detecting corrupt canaries, low values of p increase the number of runs (though not the number of *failures*) required to isolate overflows. However, lower values of p increase the precision of dangling pointer isolation by reducing the risk that certain allocation sites will always observe one canary value. We currently set $p = 1/2$, though some dangling pointer errors may require lower values of p to converge within a reasonable number of runs.

Exterminator then estimates the required lifetime extension by locating the oldest canaried object from an identified allocation site, and computing the number of allocations between the time it was freed and the time that the program failed. The correcting allocator then extends the lifetime of all objects corresponding to this allocation/deallocation site by twice this number.

5.4 Error Correction

We now describe how Exterminator uses the information from its error isolation algorithms to correct specific errors. Exterminator first generates runtime patches for each error. It then relies on a correcting allocator that uses this information, padding allocations to prevent overflows, and deferring deallocations to prevent dangling pointer errors.

```
void * correcting_malloc (size_t sz)
  // Update the allocation clock.
  clock++;
  // Free deferred objects.
  while (deferralQ.top()->time <= clock)
    really_free (deferralQ().pop()->ptr);
  int allocSite = computeAllocSite();
  // Find the pad for this site.
  int pad = padTable (allocSite);
  void * ptr = really_malloc (sz + pad);
  // Store object info and return.
  setObjectId (ptr, clock);
  setAllocSite (ptr, allocSite);
  return ptr;
```

```
void correcting_free (void * ptr)
  // Compute site info for this pointer.
  int allocS = getAllocSite (ptr);
  int freeS = computeFreeSite();
  setFreeSite (ptr, freeS);
  // Defer or free?
  int defer = deferralMap (allocS, freeS);
  if (defer == 0)
    really_free (ptr);
  else
    deferralQ.push (ptr, clock + defer);
```

Figure 5.6. Pseudo-code for the correcting memory allocator, which incorporates the runtime patches generated by the error isolator.

5.4.1 Buffer overflow correction

For every culprit-victim pair that Exterminator encounters, it generates a runtime patch consisting of the allocation site hash and the padding needed to contain the overflow (δ + the size of the overflow). If a runtime patch has already been generated for a given allocation site, Exterminator uses the maximum padding value encountered so far.

5.4.2 Dangling pointer correction

The runtime patch for a dangling pointer consists of the combination of its allocation site info and a time by which to delay its deallocation.

Exterminator computes this delay as follows. Let τ be the recorded deallocation time of the dangled object, and T be the last allocation time. Exterminator has no way of knowing how long the object is supposed to live, so computing an exact delay time is impossible. Instead, it extends the object's lifetime (delays its free) by twice the distance between its premature free and the last allocation time, plus one: $2 \times (T - \tau) + 1$.

This choice ensures that Exterminator will compute a correct patch in a logarithmic number of executions. As we show in Section 5.5.2, multiple iterations to correct pointer errors are rare in practice, because the last allocation time can be well past the time that the object should have been freed.

It is important to note that this deallocation deferral does not multiply its lifetime but rather its *drag* [65]. To illustrate, an object might live for 1000 allocations and then be freed just 10 allocations too soon. If the program immediately crashes, Exterminator will extend its lifetime by 21 allocations, increasing its lifetime by less than 1% (1021/1010). Section 5.5.3 evaluates the impact of both overflow and dangling pointer correction on space consumption.

5.4.3 The Correcting Memory Allocator

The correcting memory allocator incorporates the runtime patches described above and applies them when appropriate. Figure 5.6 presents pseudo-code for the allocation and deallocation functions.

At start-up, or upon receiving a reload signal (Section 5.1.3), the correcting allocator loads the runtime patches from a specified file. It builds two hash tables: a **pad table** mapping allocation sites to pad sizes, and a **deferral table**, mapping pairs of allocation and deallocation sites to a deferral value. Because it can reload the runtime patch file and rebuild these tables on-the-fly, Exterminator can apply patches to running programs without interrupting their execution. This aspect of Exterminator's operation may be especially useful for systems that must be kept running continuously.

On every deallocation, the correcting allocator checks to see if the object to be freed needs to be deferred. If it finds a deferral value for the object's allocation and deallocation site, it pushes onto the **deferral priority queue** the pointer and the time to actually free it (the current allocation time plus the deferral value).

The correcting allocator then checks the deferral queue on every allocation to see if an object should now be freed. It then checks whether the current allocation site has an associated pad value. If so, it adds the pad value to the allocation request, and forwards the allocation request to the underlying allocator.

5.4.4 Collaborative Correction

Each individual user of an application is likely to experience different errors. To allow an entire user community to automatically improve software reliability, Exterminator provides a simple utility that supports collaborative correction. This utility takes as input a number of runtime patch files. It then combines these patches by computing the maximum buffer pad required for any allocation site, and the maximal deferral amount for any given allocation site. The result is a new runtime patch file that covers all observed errors. Because the size of patch files is limited by the number of allocation sites in a program, we expect these files to be compact and practical to transmit. For example, the size of the runtime patches that Exterminator generates for injected errors in `espresso` was just 130K, and shrinks to 17K when compressed with gzip.

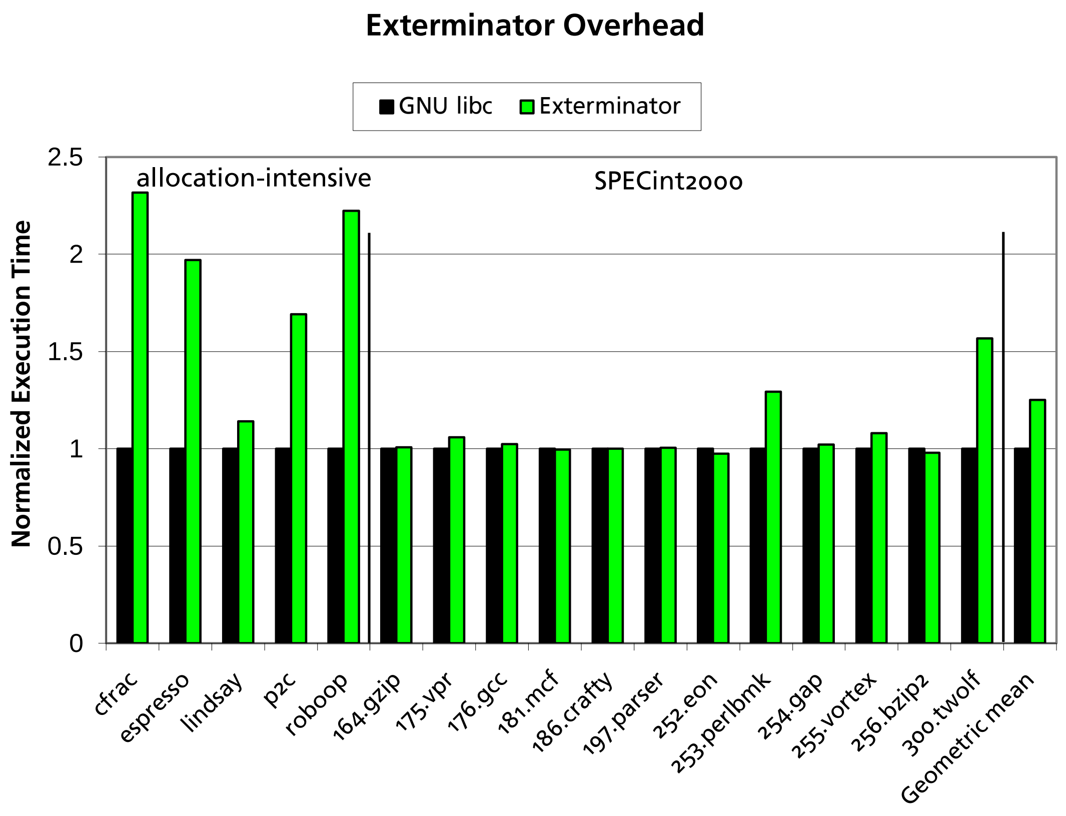


Figure 5.7. Runtime overhead for Exterminator across a suite of benchmarks, normalized to the performance of GNU libc (Linux) allocator.

5.5 Results

Our evaluation answers the following questions:

1. What is the runtime overhead of using Exterminator?

2. How effective is Exterminator at finding and correcting memory errors, both for injected and real faults?

3. What is the overhead of Exterminator's runtime patches?

5.5.1 Exterminator Runtime Overhead

We evaluate Exterminator's performance with the SPECint2000 suite [76] running reference workloads, as well as a suite of allocation-intensive benchmarks. We use the latter suite of benchmarks both because they are widely used in memory management studies [9, 31, 37], and because their high allocation-intensity stresses memory management performance. For all experiments, we fix Exterminator's heap multiplier (value of M) at 2.

All results are the average of five runs on a quiescent, dual-processor Linux system with 3 GB of RAM, with each 3.06GHz Intel Xeon processor (hyperthreading active) equipped with 512K L2 caches. Our observed experimental variance is below 1%.

We focus on the non-replicated mode (iterative/cumulative), which we expect to be a key limiting factor for Exterminator's performance and the most common usage scenario.

We compare the runtime of Exterminator (DieFast plus the correcting allocator) to the GNU libc allocator. This allocator is based on the Lea allocator [44], which is among the fastest available [11]. Figure 5.7 shows that, versus this allocator, Exterminator degrades performance by from 0% (`186.crafty`) to 132% (`cfrac`), with a geometric mean of 25.1%. While Exterminator's overhead is substantial for the allocation-intensive suite (geometric mean: 81.2%), where the cost of computing allocation and deallocation contexts dominates, its overhead is significantly less pronounced across the SPEC benchmarks (geometric mean: 7.2%).

5.5.2 Memory Error Correction

5.5.2.1 Injected Faults

To measure Exterminator's effectiveness at isolating and correcting bugs, we used the fault injector that accompanies the DieHard distribution to inject buffer overflows and dangling pointer errors. For each data point, we run the injector using a random seed until it triggers an error or divergent output. We next use this seed to deterministically trigger a single error in Exterminator, which we run in iterative mode. We then measure the number of iterations required to isolate and generate an appropriate runtime patch. The total number of images (iterations plus the first run) corresponds to the number of replicas that would be required when running Exterminator in replicated mode.

Buffer overflows: We triggered 10 different buffer overflows each of three different sizes (4, 20, and 36 bytes) by underflowing objects in the `espresso` benchmark. The number of images required to isolate and correct these errors was 3 in every case. Notice that this result is substantially better than the analytical worst-case. For three images, Theorem 3 bounds the worst-case likelihood of missing an overflow to 42% (Section 5.2.1), rather than the 0% false negative rate we observe here.

Dangling pointer errors: We then triggered 10 dangling pointer faults in `espresso` with Exterminator running in iterative and in cumulative modes. In iterative mode, Exterminator succeeds in isolating the error in only 4 runs. In another 4 runs, `espresso` does not write through the dangling pointer. Instead, it reads a canary value through the dangled pointer, treats it as valid data, and either crashes or aborts. Since no corruption is present in the heap, Exterminator cannot isolate the source of the error. In the remaining 2 runs, writing canaries into the dangled object triggers a cascade of errors that corrupt large segments of the heap. In these cases, the corruption destroys the information Exterminator requires to isolate the error.

In cumulative mode, however, Exterminator successfully isolates all 10 injected errors. For runs where no large-scale heap corruption occurs, Exterminator requires between 22 and 30 executions to isolate and correct the errors. In each case, 15 failures must be observed before the erroneous site pair crosses the likelihood threshold. Because objects are overwritten randomly, the number of runs required to yield 15 failures varies. Where writing canaries corrupts a large fraction of the heap, Exterminator requires 18 failures and 34 total runs. In some of the runs, execution continues long enough for the allocator to reuse the culprit object, preventing Exterminator from observing that it was overwritten.

5.5.2.2 Real Faults

We also tested Exterminator with actual bugs in two applications: the Squid web caching server, and the Mozilla web browser.

Squid web cache

Version 2.3s5 of Squid has a buffer overflow; certain inputs cause Squid to crash with either the GNU libc allocator or the Boehm-Demers-Weiser collector [9, 59].

We run Squid three times under Exterminator in iterative mode with an input that triggers a buffer overflow. Exterminator continues executing correctly in each run, but the overflow corrupts a canary. Exterminator's error isolation algorithm identifies a single allocation site as the culprit and generates a pad of exactly 6 bytes, fixing the error.

Mozilla web browser

We also tested Exterminator's cumulative mode on a known heap overflow in Mozilla 1.7.3 / Firefox 1.0.6 and earlier. This overflow (Bugzilla ID 307259) occurs because of an error in Mozilla's processing of Unicode characters in domain names. Not only is Mozilla multi-threaded, leading to non-deterministic allocation behavior, but even slight differences in moving the mouse cause allocation sequences to diverge. Thus, neither replicated nor iterative modes can identify equivalent objects across multiple runs.

We perform two case studies that represent plausible scenarios for using Exterminator's cumulative mode. In the first study, the user starts Mozilla and immediately loads a page that triggers the error. This scenario corresponds to a testing environment where a proof-of-concept input is available. In the second study, the user first navigates through a selection of pages (different on each run), and then visits the error-triggering page. This scenario approximates deployed use where the error is triggered in the wild.

In both cases, Exterminator correctly identifies the overflow with no false positives. In the first case, Exterminator requires 23 runs to isolate the error. In the second, it requires 34 runs. We believe that this scenario requires more runs because the site that produces the overflowed object allocates more correct objects, making it harder to identify it as erroneous.

5.5.3 Patch Overhead

Exterminator's approach to correcting memory errors does not impose additional execution time overhead in the presence of patches. However, it consumes additional space, either by padding allocations or by deferring deallocations. We measure the space overhead for buffer overflow corrections by multiplying the size of the pad by the maximum number of live objects that Exterminator patches. The most space overhead we observe is for the buffer overflow experiment with overflows of size 36, where the total increased space overhead is between 320 and 2816 bytes.

We measure space overhead for dangling pointer corrections by multiplying the object size by the number of allocations for which the object is deferred; that is, we compute the total additional drag. In the dangling pointer experiment, the amount of excess memory ranges from 32 bytes to 1024 bytes (one 256 byte object is deferred for 4 deallocations). This amount constitutes less than 1% of the maximum memory consumed by the application.

5.6 Conclusion

Like our previous systems, Exterminator operates entirely at the runtime level on unaltered binaries, and consists of three key components: (1) DieFast, a probabilistic debugging allocator, (2) a probabilistic error isolation algorithm, and (3) a correcting memory allocator. Exterminator's probabilistic error isolation isolates the source and extent of memory errors with provably low false positive and false negative rates.

Its correcting memory allocator incorporates runtime patches that the error isolation algorithm generates to correct memory errors. While Exterminator is valuable for use during testing, a key advantage over existing systems is its low overhead that allows it to protect deployed applications from crashes and security vulnerabilities by automatically discovering and correcting memory errors.

CHAPTER 6
RELATED WORK

The problem of coping with software errors has been studied extensively. In this chapter, we discuss related systems for coping with memory errors, as well as techniques for dealing with other types of software errors.

This chapter is organized as follows. First, we cover related work on coping with memory errors. Section 6.1 discusses techniques for coping with buffer overflow errors, followed by Section 6.2 on dangling pointer errors. Section 6.3 discusses allocator techniques specifically targeted at improving security, including randomization. Section 6.4 discusses previous work not directly related to memory errors, including other uses of virtual memory for user-space memory management (Section 6.4.1), and automatic techniques for repairing (Section 6.4.2) and isolating (Section 6.4.3) general software errors.

6.1 Buffer Overflows

Many previous systems have addressed the problem of buffer overflows using modified versions of C and some combination of static analysis and dynamic checks. Systems reliant on language extensions include Cyclone [36, 77], which augments C with an advanced type system to provide safe explicit memory management. CCured [52] inserts dynamic checks into the compiled program and uses static analysis to eliminate checks from places where memory errors cannot occur.

Other approaches are less ambitious, but do not require source code modification. Jones and Kelley's system [39] extends GCC to maintain metadata on all memory blocks and adds dynamic checks on each object access to detect dereferences of out-of-bounds pointers. Upon detecting an invalid dereference, the runtime prints an error message and aborts the program, preventing security vulnerabilities. While it maintains backwards compatibility for programs which maintain strict compliance with the C specification, many programs create invalid intermediate pointers which are never dereferenced, but violate the specification and create false positives in their system. Programs instrumented with their system suffer around 12X performance degradation, making it impractical for most production systems.

CRED [68] extends Jones and Kelley's scheme to handle invalid intermediate pointers, and targets only references to string buffers, which are the most common cause of buffer overflow security vulnerabilities. Dhurjati and Adve present a similar system built in LLVM [43] which uses pool allocation [42] and pointer analysis to remove many dynamic checks [26]. Their system achieves low overhead but requires whole-program analysis, making it impractical for many environments.

Rinard et al. extend CRED to *tolerate* buffer overflow errors using a system called *boundless buffers* that caches out-of-bound writes in a hash table for later reuse [63]. This approach allows the program to continue past the error and maintain correct execution in most cases.

Libraries like LibSafe and HeapShield can prevent overflows that stem from misuse of C APIs like `strcpy` [4, 7]. HeapShield itself was integrated into DieHard [10] and has also been integrated into DieHarder.

Buffer overflow detection has been offered by dynamic binary instrumentation tools used for bug detection, such as Purify [34] and Valgrind [53, 54]. These systems have high precision but very high overhead, making them unsuitable for production use.

Finally, there have been numerous debugging memory allocators which have varying support for detecting buffer overflows. The documentation for one of them, *mpa-*

trol, includes a list of over ninety such systems [67]. Notable recent allocators with debugging features include dnmalloc [84], Heap Server [41], and version 2.8 of the Lea allocator [44, 64]. Electric Fence [57] and PageHeap [49] are both object-per-page allocators like Archipelago, but without Archipelago's features that reduce its overhead.

6.2 Dangling Pointers

Conservative garbage collection [16] can prevent dangling pointer errors and some memory leaks by ignoring all `free` operations and instead automatically reclaiming unreachable memory. Despite its benefits, however, practical collectors for C/C++ have real and perceived drawbacks, both for performance and correctness. Pointer misidentification can cause conservative garbage collectors to fail to reclaim memory, especially on 32-bit platforms [15]. Worse, because C/C++ programs can obscure pointers (e.g., via XOR-encoding of linked lists [80]), a conservative collector can inadvertently reclaim live objects, causing these programs to crash.

Object-per-page allocators such as Electric Fence [57] and PageHeap [49] can detect dangling pointer accesses by remembering which pages were used for freed objects. However, these systems require substantial extra physical memory, making them suitable only for debugging.

Dhurjati and Adve improve on this technique by proposing a novel object-per-virtual-page allocator [25]. Their allocator maps multiple virtual pages to a single physical one, thus eliminating extra physical memory overhead. Upon `free`, the allocator protects the virtual page used by the object so that future (invalid) accesses can be detected. The system *soundly* reuses virtual addresses by removing the protected page entries when a whole-program pointer and liveness analysis proves that the pages are inaccessible.

6.3 Allocator Security

Other previous work to increase the security of memory allocators has focused on securing heap metadata and the use of randomization to increase non-determinism.

6.3.1 Metadata Protection

One approach is to secure the metadata via encryption: Robertson describes the use of XOR-encoded heap metadata [64], a countermeasure that was incorporated (in slightly modified form) by Lea into DLmalloc version 2.8 (a later version than the basis of GNU libc's allocator). Younan et al. instead present a modified version of the Lea allocator that fully segregates metadata, but which implements no other security enhancements [84]. Kharbutli et al. describe an approach to securing heap metadata that places it in a separate process [41]. Isolation of heap metadata helps prevent certain attacks but, for example, does not mitigate attacks against the heap data itself. Like DieHard, DieHarder completely segregates heap metadata, and its randomized placement of heap metadata in a sparse address space effectively protects the metadata.

6.3.2 Randomized Memory Managers

Several memory management systems employ some degree of randomization, including locating the heap at a random base address [13, 56], adding random padding to allocated objects [14], shuffling recently-freed objects [41], or a mix of padding and object deferral [59]. This level of randomization is insufficient for Exterminator, which requires full heap randomization. None of these approaches generate as much entropy as DieHard or DieHarder.

Exterminator builds on DieHard [9], which tolerates errors probabilistically. Exterminator substantially modifies and extends DieHard's heap layout and allocation algorithms. It also uses probabilistic algorithms that identify and correct errors.

6.3.3 Heap Spraying Countermeasures

One noteworthy countermeasure by Ratanaworabhan et al. called Nozzle addresses heap spraying attacks aimed at preventing code injection attacks [61]. Nozzle operates by scanning the heap looking for valid x86 code sequences—a large number of such sequences indicates that a spray attack is in progress, and can be used to trigger program termination.

6.4 Other Related Work

6.4.1 VM techniques for memory management

Recent cooperative systems exploit communication between the OS virtual memory manager (VMM) and the garbage collector to reduce paging due to garbage collection. Yang et al. modify the Linux virtual memory manager to provide detailed reference information, allowing it to dynamically adapt the GC heap size in order to maximize performance [83].

Appel and Li describe a number of primitives and algorithms for exploiting virtual memory in user-mode [3].

6.4.2 Automatic Repair

Demsky et al.'s *automatic data structure repair* [21, 22, 23] enforces data structure consistency specifications, guided by a formal description of the program's data structures (specified manually or derived automatically by Daikon [27]). Exterminator attacks a different problem, namely that of isolating and correcting memory errors, and is orthogonal and complementary to data structure repair.

Sidiroglou et al. propose STEM, a self-healing runtime that executes functions in a transactional environment so that if they detect the function misbehaving, they can prevent it from doing damage [71]. Using STEM, they implement error virtualization, which maps the set of possible errors in a function onto those that have an explicit error handler. The more recent SEAD system goes beyond STEM requiring no source code changes, handling I/O with virtual proxies, and by specifying the repair policy explicitly through an external description [72]. While STEM and SEAD are promising approaches to automatically recovering from errors, neither provides solutions for as broad a class of errors as Exterminator, nor do they provide mechanisms to semantically eliminate the source of the error automatically, as Exterminator does.

6.4.3 Automatic Debugging

Two previous systems apply techniques designed to help isolate bugs. *Statistical bug isolation* is a distributed assertion sampling technique that helps pinpoint the location of errors, including but not limited to memory errors [45, 46, 47]. It works by injecting lightweight tests into the source code; the result of these tests, in bit vector form, can be processed to generate likely sources of the errors. This statistical processing differs from Exterminator's probabilistic error isolation algorithms, although Liu et al. also use hypothesis testing [47]. Like statistical bug isolation, Exterminator can leverage runs of deployed programs. However, unlike statistical bug isolation, Exterminator requires neither source code nor a large deployed user base in order to find errors, and automatically generates runtime patches that correct them.

Delta debugging automates the process of identifying the smallest possible inputs that do and do not exhibit a given error [19, 50, 85]. Given these inputs, it is up to the software developer to actually locate the bugs themselves. Exterminator focuses on a narrower class of errors, but is able to isolate and correct an error given just one erroneous input, regardless of its size.

6.4.4 Fault Tolerance

Recently, there has been an increasing focus on approaches for tolerating hardware transient errors that are becoming more common due to fabrication process limitations. Work in this area ranges from proposed hardware support [60] to software fault tolerance [62]. While Exterminator also uses redundancy as a method for detecting and correcting errors, Exterminator goes beyond tolerating software errors, which are not transient, to correcting them permanently. Like Exterminator, other efforts in the fault tolerance community seek to gather data from multiple program executions to identify potential errors. For example, Guo et al. use statistical techniques on internal monitoring data to probabilistically detect faults, including memory leaks and deadlocks [32]. Exterminator goes beyond this previous work by characterizing each memory error so specifically that a correction can be automatically generated for it.

Rx operates by checkpointing program execution and logging inputs [59]. Rx rolls back crashed applications and replays inputs to it in a new environment that pads all allocations or defers all deallocations by some amount. If this new environment does not yield success, Rx rolls back the application again and increases the pad values, up to some threshold. Unlike Rx, Exterminator does not require checkpointing or rollback, and precisely isolates and corrects memory errors.

CHAPTER 7
CONCLUSION

Despite years of research and dozens of tools, memory errors remain a problem in deployed software. Existing debugging tools succeed at finding these errors, but require high runtime or memory overheads, making them unsuitable for deployment. Many such errors may manifest only after long periods in production. Others create security risks when used with uncontrolled inputs such as web pages or arbitrary network requests. Runtime systems that tolerate memory errors and prevent their exploitation decrease these risks.

7.1 Contributions

In this thesis, we present three systems that improve application reliability and security in the presence of memory errors. First, it presents Archipelago, which tolerates large and repeated heap overflows in especially-vulnerable server applications. We show that applications using Archipelago can survive thousands of repeated memory errors without malfunctioning. Second, it presents DieHarder, an allocator specifically designed to reduce predictability and to make exploiting attacks as difficult as possible while remaining practical. We show that DieHarder provides significantly more entropy than previous allocators while imposing little runtime overhead. Finally, it presents Exterminator, a system that uses robust statistical techniques to tolerate, detect, and correct heap buffer overflows and dangling pointers in general-purpose applications. We show that Exterminator has acceptably-low runtime overhead for production use as well as provably low false positive and negative rates.

7.2 Future Work

The systems presented in this thesis prove that the underlying concepts are sound, while suggesting enhancements and further exploration of these ideas for future work. Our current implementation of Archipelago uses a fixed-size hot space, that is, a constant number of objects are kept uncompressed and directly-accessible. A more robust implementation would use an adaptively-sized space, requiring less physical memory when possible, but providing better performance for larger working sets. A mechanism such as CRAMM's [83] that dynamically adjusts the hot space size to maintain a target overhead would obviously be applicable here.

Our current implementation of DieHarder allocates small-object pages uniformly at random from a large, statically-sized virtual address space. While this policy maximizes entropy, it can require significant memory to store page tables. OpenBSD's kernel allocator uses a more adaptive mechanism that allocates randomly from a virtual address region that grows as more pages become allocated. Adopting such a mechanism for DieHarder (on Linux) would reduce memory overhead while maintaining $O(\log N)$ bits of entropy in page addresses.

Exterminator's existing implementation focuses on reliability, rather than security. While the existing system provides significant benefits, minor changes could substantially improve its adaptation to attacks. For example, upon discovering an allocation site responsible for overflowed objects, it could use Archipelago to allocate those objects on their own pages, surrounded by guard pages. This policy would ensure that contiguous overflows from those objects could never overwrite vulnerable data.

Exterminator's cumulative mode error isolation algorithms, which require neither repeated runs on the same input nor determinism, are the most applicable for production use. However, they may require many iterations (around 30) in order to diagnose the error. Replacing DieFast's heap layout with DieHarder's, which consists of many almost-independent pages rather than large contiguous regions of address space, would greatly decrease the number of iterations required for Exterminator to identify the cause of an error.

Different inference methods that use richer information collected from individual heap dumps may also enable quicker convergence. For example, for buffer overflows, the algorithm keeps only a single bit per allocation site representing whether an object from that callsite existed at a lower address in the same miniheap as a corrupted canary. Tracking more information, such as possible deltas, would provide the isolator with much more information without requiring excessive space, and would likely enable more quickly-converging inference algorithms.

BIBLIOGRAPHY

[1] Jonathan Afek and Adi Sharabani. Dangling pointer: Smashing the pointer for fun and profit. In *Black Hat USA* (2007).

[2] Alexander Anisimov. Defeating Microsoft Windows XP SP2 heap protection and DEP bypass, 2005.

[3] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *ASPLOS* (1991), pp. 96–107.

[4] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Tied, libsafeplus: Tools for runtime buffer overflow protection. In *Proceedings of the 13th USENIX Security Symposium* (Aug. 2004), USENIX.

[5] BBP. BSD heap smashing. `http://www.ouah.org/BSD-heap-smashing.txt`.

[6] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (2010), ASPLOS '10, ACM.

[7] Emery D. Berger. HeapShield: Library-based heap overflow protection for free. Tech. Rep. UMCS TR-2006-28, Department of Computer Science, University of Massachusetts Amherst, May 2006.

[8] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for c/c++. In *OOPSLA* (2009), Shail Arora and Gary T. Leavens, Eds., ACM, pp. 81–96.

[9] Emery D. Berger and Benjamin G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *PLDI* (2006), Michael I. Schwartzbach and Thomas Ball, Eds., ACM, pp. 158–168.

[10] Emery D. Berger and Benjamin G. Zorn. Efficient probabilistic memory safety. Tech. Rep. UMCS TR-2007-17, Department of Computer Science, University of Massachusetts Amherst, Mar. 2007.

[11] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2001), ACM Press, pp. 114–124.

[12] Dan Bernstein. Usenet posting, `comp.lang.c`. `http://groups.google.com/group/comp.lang.c/msg/6b82e964887d73d9`, Dec. 1990.

[13] Sandeep Bhatkar, Daniel 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* (Aug. 2003), USENIX, pp. 105–120.

[14] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium* (Aug. 2005), USENIX, pp. 271–286.

[15] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *PLDI* (1993), pp. 197–206.

[16] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *SPE 18*, 9 (1988), 807–820.

[17] Michael D. Bond and Kathryn S. McKinley. Bell: bit-encoding online memory leak detection. In Shen and Martonosi [70], pp. 61–72.

[18] Richard W Carr and John L Hennessy. WSClock - A simple and effective algorithm for virtual memory management. In *Proceedings of the Eighth ACM Symposium on Operating System Principles* (Pacific Grove, CA, Dec. 1981), pp. 87–95.

[19] Holger Cleve and Andreas Zeller. Locating causes of program failures. In Roman et al. [66], pp. 342–351.

[20] Matt Conover and the w00w00 Security Team. w00w00 on heap overflows. http://www.w00w00.org/files/articles/heaptut.txt, January 1999.

[21] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA* (2006), Lori L. Pollock and Mauro Pezzè, Eds., ACM, pp. 233–244.

[22] Brian Demsky and Martin C. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA* (2003), Ron Crocker and Guy L. Steele Jr., Eds., ACM, pp. 78–95.

[23] Brian Demsky and Martin C. Rinard. Data structure repair using goal-directed reasoning. In Roman et al. [66], pp. 176–185.

[24] Peter Denning. Working sets past and present. 64–84.

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

[26] Dinakar Dhurjati and Vikram S. Adve. Backwards-compatible array bounds checking for C with very low overhead. In Osterweil et al. [55], pp. 162–171.

[27] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE* (2000), pp. 449–458.

[28] Justin N. Ferguson. Understanding the heap by breaking it. In *Black Hat USA* (2007).

[29] Jeanne Ferrante and Kathryn S. McKinley, Eds. *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007* (2007), ACM.

[30] Shashank Gonchigar. Ani vulnerability: History repeats.

[31] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation* (Albuquerque, NM, June 1993), vol. 28(6), pp. 177–186.

[32] Zhen Guo, Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. In *DSN* (2006), IEEE Computer Society, pp. 259–268.

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

[34] Reed Hastings and Bob Joyce. Fast detection of memory leaks and access errors. In *Proceedings of the Winter '92 USENIX conference* (1992), USENIX Association, pp. 125–136.

[35] huku. Exploiting dlmalloc frees in 2009. *Phrack 13*, 66 (November 2009).

[36] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, General Track* (2002), Carla Schlatter Ellis, Ed., USENIX, pp. 275–288.

[37] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *OOPSLA '97 Workshop on Garbage Collection and Memory Management* (Oct. 1997), Peter Dickman and Paul R. Wilson, Eds.

[38] Richard E. Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Chichester, July 1996.

[39] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG* (1997), pp. 13–26.

[40] Poul-Henning Kamp. Malloc(3) revisited. `http://phk.freebsd.dk/pubs/malloc.pdf`.

[41] Mazen Kharbutli, Xiaowei Jiang, Yan Solihin, Guru Venkataramani, and Milos Prvulovic. Comprehensively and efficiently protecting the heap. In Shen and Martonosi [70], pp. 207–218.

[42] Chris Lattner and Vikram Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), pp. 129–142.

[43] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO* (3 2004), IEEE Computer Society, pp. 75–88.

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

[45] Ben Liblit, Alexander Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI* (2003), ACM, pp. 141–154.

[46] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *PLDI* (2005), Vivek Sarkar and Mary W. Hall, Eds., ACM, pp. 15–26.

[47] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. SOBER: statistical model-based bug localization. In *ESEC/SIGSOFT FSE* (2005), Michel Wermelinger and Harald Gall, Eds., ACM, pp. 286–295.

[48] John McDonald and Chris Valasek. Practical Windows XP/2003 Heap Exploitation. In *BlackHat USA 2009* (Las Vegas, NV, July 2009).

[49] Microsoft. How to use Pageheap.exe in Windows xp, Windows 2000, and Windows Server 2003. http://support.microsoft.com/kb/286470.

[50] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical delta debugging. In Osterweil et al. [55], pp. 142–151.

[51] Otto Moerbeek. A new malloc(3) for OpenBSD. In *EuroBSDCon* (2009).

[52] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst. 27*, 3 (2005), 477–526.

[53] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. In *SPACE 2004* (Venice, Italy, Jan. 2004).

[54] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Ferrante and McKinley [29], pp. 89–100.

[55] Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, Eds. *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006* (2006), ACM.

[56] PaX Team. PaX address space layout randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt`.

[57] Bruce Perens. Electric fence `malloc` debugger. http://perens.com/FreeSoftware/ElectricFence/.

[58] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. Automatically patching errors in deployed software. In *SOSP* (2009), Jeanna Neefe Matthews and Thomas E. Anderson, Eds., ACM, pp. 87–102.

[59] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In *SOSP* (2005), Andrew Herbert and Kenneth P. Birman, Eds., ACM, pp. 235–248.

[60] Moinuddin K. Qureshi, Onur Mutlu, and Yale N. Patt. Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors. In *DSN* (2005), IEEE Computer Society, pp. 434–443.

[61] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th USENIX Security Symposium* (Aug. 2009), USENIX, pp. 169–186.

[62] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *CGO* (2005), IEEE Computer Society, pp. 243–254.

[63] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, and Tudor Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *ACSAC* (2004), IEEE Computer Society, pp. 82–90.

[64] William K. Robertson, Christopher Krügel, Darren Mutz, and Fredrik Valeur. Run-time detection of heap-based overflows. In *LISA* (2003), USENIX, pp. 51–60.

[65] Niklas Rojemo and Colin Runciman. Lag, drag, void and use — heap profiling and space-efficient compilation revisited. In *Proceedings of the first ACM SIGPLAN International Conference on Functional Programming* (1996), ACM, pp. 34–41.

[66] Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, Eds. *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA* (2005), ACM.

[67] Graeme S. Roy. mpatrol: Related software. `http://www.cbmamiga.demon.co.uk/mpatrol/mpatrol_83.html`, Nov. 2006.

[68] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *NDSS* (2004), The Internet Society.

[69] Hovav Shacham, Matthew Page, Ben Pfaff, Eu Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security* (2004).

[70] John Paul Shen and Margaret Martonosi, Eds. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006* (2006), ACM.

[71] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a reactive immune system for software services. In *USENIX Annual Technical Conference, General Track* (2005), USENIX, pp. 149–161.

[72] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. From STEM to SEAD: Speculative execution for automated defense. In *USENIX Annual Technical Conference* (2007), USENIX.

[73] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM 28* (February 1985), 202–208.

[74] Solar Designer. JPEG COM marker processing vulnerability in Netscape browsers. `http://www.openwall.com/advisories/OW-002-netscape-jpeg/`, 2000.

[75] Alexander Sotirov. Heap Feng Shui in JavaScript. In *Black Hat Europe* (2007).

[76] Standard Performance Evaluation Corporation. SPEC2000. http://www.spec.org.

[77] Nikhil Swamy, Michael W. Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in Cyclone. *Sci. Comput. Program. 62*, 2 (2006), 122–144.

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

[79] Ollie Whitehouse. An analysis of address space layout randomization on Windows Vista. `http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf`, 2007.

[80] Wikipedia. XOR linked list, 2008. [Online; accessed 19-March-2008].

[81] Wikipedia. Dangling pointer — Wikipedia, the free encyclopedia, 2010. [Online; accessed 16-April-2010].

[82] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management* (Kinross, Scotland, Sept. 1995), vol. 986, pp. 1–116.

[83] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *OSDI* (2006), USENIX Association, pp. 103–116.

[84] Yves Younan, Wouter Joosen, Frank Piessens, and Hans Van den Eynden. Security of memory allocators for C and C++. Tech. Rep. CW 419, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, July 2005. Available at `http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW419.pdf`.

[85] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *ESEC / SIGSOFT FSE* (1999), Oscar Nierstrasz and Michel Lemoine, Eds., vol. 1687 of *Lecture Notes in Computer Science*, Springer, pp. 253–267.