DieHard: Efficient Probabilistic Memory Safety

EMERY D. BERGER
University of Massachusetts Amherst
and
BENJAMIN G. ZORN
Microsoft Research

Applications written in unsafe languages like C and C++ are vulnerable to memory errors such as buffer overflows, dangling pointers, and reads of uninitialized data. Such errors can lead to program crashes, security vulnerabilities, and unpredictable behavior. We present DieHard, a randomized runtime system that tolerates these errors while probabilistically maintaining soundness. DieHard uses randomization to achieve *probabilistic memory safety* by approximating an infinite-sized heap. DieHard's memory manager randomizes the location of objects in a heap that dynamically adapts to be a constant factor larger than required. In exchange for this increased space consumption and a modest degradation in performance (geometric mean 6%), DieHard both prevents heap corruption and provides probabilistic guarantees of avoiding memory errors like dangling pointers and heap buffer overflows.

For additional safety, DieHard can operate in a *replicated* mode where multiple replicas of the same application are run simultaneously. By initializing each replica with a different random seed and requiring agreement on output, this replicated version of DieHard increases the likelihood of correct execution because errors are unlikely to have the same effect across all replicas. We present analytical and experimental results that show DieHard's resilience to a wide range of memory errors, and report on a broad deployment of DieHard to the general public.

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

General Terms: Algorithms, Languages, Reliability

1. INTRODUCTION

Most software applications are written in C and C++, two unsafe languages. These languages let programmers maximize performance but are error-prone. Memory management errors, which dominate recent security vulnerabilities reported by CERT [US-CERT], are

Emery Berger was supported by NSF CAREER Award CNS-0347339 and CNS-0615211, a grant from Intel Corporation, and a gift from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other sponsors.

Authors' addresses: E. Berger, Department of Computer Science, University of Massachusetts Amherst, Amherst, MA 01003; email: emery@cs.umass.edu; B. Zorn, Microsoft Research, One Microsoft Way, Redmond, WA 98052; email: zorn@microsoft.com.

especially pernicious. These errors fall into the following categories:

Dangling pointers. If a program mistakenly frees a live object, the allocator may overwrite its contents with a new object or heap metadata.

Buffer overflows. Out-of-bound writes can corrupt live objects on the heap.

Heap corruption. An out-of-bound write can corrupt any heap metadata stored near heap objects.

Uninitialized reads. Reading values from newly-allocated or unallocated memory leads to undefined behavior.

Invalid frees. Passing illegal addresses to `free` can corrupt the heap or lead to undefined behavior.

Double frees. Repeated calls to `free` of objects that have already been freed cause freelist-based allocators to fail.

Tools like Purify [Hastings and Joyce 1991] and memgrind (part of Valgrind) [Nethercote and Seward 2007; Nethercote and Fitzhardinge 2004; Seward and Nethercote 2005] allow programmers to pinpoint the exact location of these memory errors (at the cost of a 2-25X performance penalty), but only reveal those bugs found during testing. Deployed programs thus remain vulnerable to crashes or attack. Conservative garbage collectors can, at the cost of increased runtime and additional memory [Detlefs 1993; Hertz and Berger 2005], disable calls to `free` and eliminate three of the above errors (invalid frees, double frees, and dangling pointers). Assuming source code is available, a programmer can also compile the code with a safe C compiler that inserts dynamic checks for the remaining errors, further increasing running time [Austin et al. 1994; Avots et al. 2005; Dhurjati and Adve 2006; Dhurjati et al. 2006; Necula et al. 2002; Xu et al. 2004; Yong and Horwitz 2003]. As soon as an error is detected, the inserted code aborts the program.

While this fail-stop approach is safe, aborting a computation is often undesirable, since users are rarely happy to see their programs suddenly stop. In order to prolong execution in the face of memory errors, some systems sacrifice soundness [Qin et al. 2005; Rinard et al. 2004]. For example, failure-oblivious computing builds on a safe C compiler but drops (or caches) illegal writes and manufactures values for invalid reads. Unfortunately, these systems provide no assurance to programmers that their programs are executing correctly.

This article makes the following contributions:

(1) It introduces the notion of **probabilistic memory safety**, a probabilistic guarantee of avoiding memory errors.

(2) It presents **DieHard**, a runtime system that provides probabilistic memory safety. We show analytically and empirically that DieHard eliminates or avoids all of the memory errors described above with high probability.

This article builds on our prior work [Berger and Zorn 2006], which introduced DieHard. The most important change is a new, adaptive randomized memory management algorithm that forms the core of DieHard (Section 4). The original algorithm required an *a priori* static heap size that was at least 24 times larger than the maximum amount required for any object size. This large virtual memory requirement severely limited the range of applications that could run with DieHard in 32-bit address spaces.

By contrast, the adaptive algorithm presented here adjusts to application memory usage and increases memory consumption only by the required heap expansion factor M, where

$M \geq 1$ (e.g., $3/2$ or 2). This algorithm thus dramatically reduces space consumption and runtime overhead by reducing TLB and L2 misses (Section 7.2), while preserving the probabilistic memory safety guarantees of the original algorithm.

1.1 Outline

The remainder of this article is organized as follows. Section 2 provides an overview of how DieHard provides probabilistic memory safety. Section 3 formalizes the notions of probabilistic memory safety and introduces infinite-heap semantics, which probabilistic memory safety approximates. Section 4 then presents DieHard's fast, randomized memory allocator that forms the heart of the stand-alone and replicated versions. Section 5 describes DieHard's replicated variant. Section 6 presents analytical results for both versions, and Section 7 provides empirical results, measuring overhead and demonstrating DieHard's ability to avoid memory errors. Sections 8 discusses related work, and Section 9 concludes with a discussion of future directions.

2. OVERVIEW

DieHard provides two modes of operation: a **stand-alone** mode that replaces the default memory manager, and a **replicated** mode that runs several replicas simultaneously. Both rely on a novel **randomized memory manager** that allows the computation of the exact probabilities of detecting or avoiding memory errors.

DieHard uses a bitmap-based, fully-randomized memory manager to provide *probabilistic memory safety*. It allocates from a heap sized M times larger than the maximum needed for the application. Allocation randomly probes the bitmaps associated with the given size class for a free bit: this operation takes $O(1)$ expected time (see Section 4.2). 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. DieHard also improves application robustness by segregating all heap metadata from the heap (avoiding most heap metadata overwrites) and ignoring attempts to `free` already-freed objects. Despite its degradation of spatial locality, we show that the DieHard memory manager's impact on performance is small for many applications (average 6% across the SPECint2000 benchmark suite).

While the stand-alone version of DieHard provides substantial protection against memory errors, DieHard can optionally use replication (Figure 1) to further increase the probability of successful execution in the face of errors. It broadcasts inputs to a number of replicas, each of which is a copy of the application process 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. With high probability, whenever any two programs agree on their output, they have executed safely. In other words, in any agreeing replicas, any buffer overflows only overwrote empty space or dead data, and dangling pointers were never overwritten. 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. In addition, if an application's output depends on uninitialized data, because these data will be different across the replicas, DieHard will detect them.

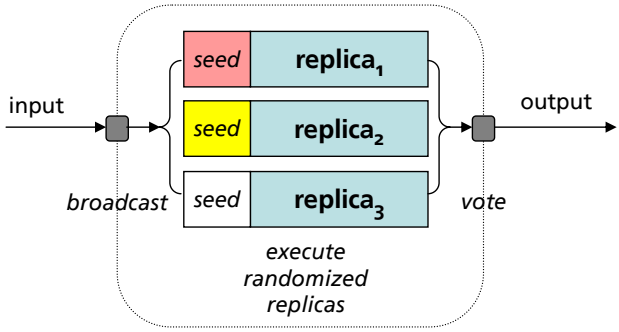


Fig. 1. The replicated DieHard architecture. Input is broadcast to multiple replicas, each equipped with a different, fully-randomized memory manager. Output is only committed when at least two replicas agree on the result.

Since replacing the heap with DieHard significantly improves reliability, it is suitable for broad deployment, especially in scenarios where increased reliability is worth the space cost. For example, a buggy version of the Squid web caching server crashes on ill-formed inputs when linked with both the default GNU libc allocator and the Boehm-Demers-Weiser garbage collector, but runs correctly with DieHard (see Section 7.3.2). Section 7.3.2 also describes our experience with a broad public deployment tailored to the Mozilla Firefox web browser.

Using additional replicas further increases reliability. While additional replicas would naturally increase execution time on uniprocessor platforms, we believe that the natural setting for using replication is on systems with multiple processors. It has proven difficult to rewrite applications to take advantage of multiple CPUs in order to make them run faster. DieHard can instead use the multiple cores on newer processors to make legacy programs more reliable.

3. PROBABILISTIC MEMORY SAFETY

We define a program as being **fully memory safe** if it satisifies the following criteria: it never reads uninitialized memory, performs no illegal operations on the heap (no invalid/-double frees), and does not access freed memory (no dangling pointer errors).

By aborting a computation that might violate one of these conditions, a safe C compiler provides full memory safety. However, an ideal execution environment would allow such programs to continue to execute correctly (soundly) in the face of many of these errors.

We can define such an idealized, but unrealizable, runtime system. We call this runtime system an **infinite-heap memory manager**, and say that it provides **infinite-heap semantics**. In such a system, the heap area is infinitely large, so there is no risk of heap exhaustion. Objects are never deallocated, and all objects are allocated infinitely far apart from each other (that is, they can be thought of as *boundless memory blocks* [Rinard et al. 2004]).

From the standpoint of a correct C execution, a program that does not deliberately seek to exhaust the heap cannot tell whether it is running with an ordinary heap implementation or an infinite heap. However, infinite-heap semantics allows programs to execute safely that would be rejected by a safe C compiler. Because every object is infinitely far from every other object, heap buffer overflows are benign — they never overwrite live data. The

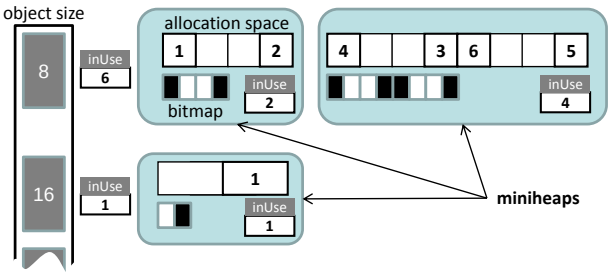


Fig. 2. The adaptive (new) DieHard heap layout. Objects in the same size class are allocated randomly from separate regions ("miniheaps"), each of which holds M times more memory than required (here, $M = 2$).

problems of heap corruption and dangling pointers also vanish because frees are ignored and allocated objects are never overwritten. However, uninitialized reads to the heap remain undefined. Unlike Java, the contents of newly-allocated C and C++ objects are not necessarily defined. [1]

3.1 Approximating infinite heaps

While an infinite-heap memory manager is unimplementable, we can probabilistically approximate its behavior. We replace the infinite heap with one that is M times larger than the maximum required to obtain an M-approximation to infinite-heap semantics. By placing objects uniformly at random across the heap, we get a minimum expected separation of $E[\text{minimum separation}] = M - 1$ objects, making overflows smaller than $M - 1$ objects benign. In addition, by randomizing the choice of freed objects to reclaim, prematurely-freed objects are highly unlikely to be overwritten.

3.2 Detecting uninitialized reads

This memory manager approximates most aspects of infinite-heap semantics as M approaches infinity. By filling the heap with random values, DieHard can detect reads to uninitialized objects by simultaneously executing at least two *replicas* with different randomized allocators and comparing their outputs. An uninitialized read will return different results across the replicas, and if this read affects the computation, the outputs of the replicas will differ.

4. RANDOMIZED MEMORY MANAGEMENT

This section describes the randomized memory management algorithm that approximates the infinite heap semantics given above. We first describe the heap layout, and then describe the allocation and deallocation algorithms. We use interposition to replace the allocation calls in the target application; see Section 5.1 for details.

4.1 Heap Layout

Figure 2 depicts DieHard's heap layout. While the original version of DieHard used a fixed, statically-sized heap, the version described here is adaptive: it dynamically increases the amount of memory as needed. We first describe the new, *adaptive* allocator in detail. We then briefly present the original, *static* DieHard allocation algorithm.

[1] ISO C++ Standard 5.3.4, paragraph 14A.

The adaptive version of DieHard dynamically sizes its heap to be M times larger than the maximum needed by the application, where M can be any fraction greater than or equal to 1. It allocates memory from increasingly large chunks that we call *miniheaps*. Each miniheap contains objects of exactly one size.

DieHard allocates miniheaps and objects larger than 64K directly using `mmap`, and places guard pages without read or write access on either end of these regions. Object requests are rounded up to the nearest power of two. Using powers of two speeds allocation by allowing expensive division and modulus operations to be replaced with bit-shifting. It also adds resilience to buffer overflows when objects are smaller than the next power of two, although we ignore this effect in our analyses.

Separate regions are essential to making the allocation algorithm practical. If objects were instead randomly spread across the entire heap area, fragmentation would be a certainty, because small objects would be scattered across all of the pages. Restricting each size class to its own region eliminates this external fragmentation. We discuss DieHard's memory efficiency further in Section 4.5. This organization also allows DieHard to efficiently prevent heap overflows caused by unsafe library functions like `strcpy`, as we describe in Section 4.4.

One aspect that makes this layout robust to errors and security vulnerabilities is its complete separation of heap metadata from heap objects. Many allocators, including the Lea allocator that forms the basis of the GNU libc allocator, store heap metadata in areas immediately adjacent to allocated objects ("boundary tags"). A buffer overflow of just one byte past an allocated space can corrupt the heap, leading to program crashes, unpredictable behavior, or security vulnerabilities [Kaempf 2001]. Other allocators place such metadata at the beginning of a page, reducing but not eliminating the likelihood of corruption. Keeping all of the heap metadata separate from the heap protects it from buffer overflows.

The heap metadata includes a bitmap for each miniheap, where one bit always stands for one object. All bits are initially zero, indicating that every object is free. Additionally, DieHard tracks the total number of objects allocated across all the miniheaps (`inUse`); this number is used to add a new miniheap when the number of objects would exceed the threshold factor of $1/M$.

For the replicated version only, DieHard fills the heap with random values. Each replica's random number generator is seeded with a "true" random number. For example, the Linux version reads from `/dev/urandom`, which uses various system-specific measures to compute a number that is effectively random. DieHard's random number generator is an inlined version of Marsaglia's multiply-with-carry random number generation algorithm, which is a fast, high-quality source of pseudo-random numbers [Marsaglia 1994].

4.2 Object Allocation

When an application requests memory from DieHard's `malloc`, the allocator first checks to see whether the request is for a large object (larger than 64K); if so, it allocates memory directly via `mmap`. Otherwise, it computes the size class for the allocation request ($\lceil \log_2 \rceil - 3$; the smallest allocation size is eight bytes). As long as the corresponding set of miniheaps is not already full, it then looks for space.

If an allocation request would cause the total number of objects in use in all miniheaps for a particular size class to exceed $1/M$ of allocated space, DieHard first allocates a new miniheap that is twice as large as the previous largest miniheap.

Next, DieHard allocates memory by searching for a free slot in all miniheaps in the re-

quested size class. It repeatedly chooses a random miniheap to allocate from, and checks whether a random slot within that miniheap is free. Upon finding a free slot, the allocator sets the appropriate bit in the bitmap to mark it as allocated and increments the number of objects in use. For the replicated version, it also fills the object with randomized values; DieHard relies on this randomization to detect uninitialized reads, as we describe in Section 5. Finally, the allocator returns the address corresponding to this slot.

The fact that the heap can be at most $1/M$ full bounds the expected runtime of allocation to a small constant. Each search for an unused slot is a Bernoulli trail with worst-case odds of success $p = (M-1)/M$. The expected number of attempts until one success is then $\frac{1}{1-(1/M)}$. In particular, for $M = 2$, the expected worst-case number of probes is 2.

It is important to ensure that the odds of choosing any given free object are equally likely, regardless of which miniheap they may be in. The allocator converts a random number from 1 to n into an appropriate miniheap index by computing its log base 2. This conversion makes it twice as likely that we will choose miniheap index $i+1$ over miniheap i, reflecting the fact that each miniheap is twice as large as the previous one.

4.3 Object Deallocation

To defend against erroneous programs, DieHard's `free` implementation takes several steps that ensure that any object given to it is in fact valid. First, it checks to see if the address to be freed is inside a miniheap by iterating through the miniheaps associated with each size class. If the object is inside a miniheap and currently allocated, DieHard resets the bit corresponding to the object's location in the bitmap and decrements the count of allocated objects for the given size class.

If DieHard does not find the object in any of the miniheaps or size classes, the object is either a large object (allocated with `mmap`) or it is invalid. DieHard checks a table of allocated large objects to ensure that this object was indeed returned by a previous call to `mmap`. If so, it `munmap`s the object; otherwise, it ignores the request.

When freeing an object, DieHard rounds the address down to the nearest allocatable address in that miniheap. For example, a `free` call to a pointer 4 bytes beyond the start of an object is equivalent to a `free` call to the start of the object. This policy has few disadvantages and several advantages over the alternative, which would be to ignore such `free` calls. First, if the object was unintentionally passed to free, the resulting dangling pointer error will likely have little impact due to DieHard's protection against such faults (see Section 6.1). If, however, the object was intended to be freed, ignoring it would cause a memory leak. This policy also simplifies DieHard's implementation of `memalign`, a variant of `malloc` that returns a pointer aligned to a given block size. DieHard simply returns an aligned pointer within an appropriately-sized `malloc`'d chunk. When the client invokes `free` on the aligned pointer, DieHard's rounding down causes the `malloc`'d chunk to be freed.

While this iteration through size classes and miniheaps may seem costly, it is efficient both in theory and in practice. The number of size classes a constant, and the use of power-of-two size classes makes this number small. DieHard heuristically iterates from smaller to larger size classes, taking advantage of the fact that in most programs, the most frequently-used object sizes are small.

In addition to this heuristic, DieHard's deallocator orders its iteration in a way that reduces its expected asymptotic complexity. Within each size class, the number of miniheaps is $O(\log_2 n)$, where n is the maximum number of objects allocated from that size class. Die-

Hard's deallocation routine iterates from the largest miniheap to the smallest. While the worst-case cost of DieHard's deallocation is $O(\log_2 n)$, the expected number of iterations to free a valid object is $O(1)$.

To illustrate this analysis, consider a program that has already allocated some threshold number of miniheaps, and then performs a sequence of n deallocations and allocations. The allocated objects will be randomly spread throughout the miniheaps. If $M = 2$, approximately $1/2$ of the objects will be in the largest miniheap, and these objects will be deallocated in one iteration with probability $1/2$. Likewise, $1/4$ of the objects will be in the second-largest miniheap, so the cost will be two iterations with probability $1/4$. The expected cost to deallocate any of these n objects will thus be $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$. Thus, both DieHard's allocation and deallocation routines run in expected $O(1)$ time.

4.4 Limiting Heap Buffer Overflows

Not only does DieHard's heap layout provide probabilistic protection against heap buffer overflows (see Section 6.2), it also makes it efficient to prevent overflows caused by unsafe library functions like `strcpy`. DieHard replaces these unsafe library functions with variants that do not write beyond the allocated area of heap objects. Each function first checks if the destination pointer lies within the heap using the logic described above for deallocation (expected $O(1)$ operations). If the pointer is within the heap, DieHard finds the start of the object by bitmasking the pointer with its size (computed with a bitshift) minus one. DieHard then computes the available space from the pointer to the end of the object (one subtractions). With this value limiting the maximum number of bytes to be copied, DieHard prevents `strcpy` from causing heap buffer overflows.

In addition to replacing `strcpy`, DieHard also replaces its "safe" counterpart, `strncpy`. This function requires a length argument that limits the number of bytes copied into the destination buffer. The standard C library contains a number of these checked library functions in an attempt to reduce the risk of buffer overflows. However, checked functions are little safer than their unchecked counterparts, since programmers can inadvertently specify an incorrect length. As with `strcpy`, the DieHard version of `strncpy` checks the actual available space in the destination object and uses that value as the upper bound.

We note that this approach is not specific to the DieHard allocator. We have implemented similar functionality for a modified version of the PHKmalloc allocator (used in FreeBSD) and demonstrated its speed and efficacy at avoiding library-based heap overflows [Berger 2006]. The approach could be applied to any allocator that uses a BiBoP (big bag-of-pages) allocation scheme, which permit rapid location of an object's metadata given any pointer within the object [Hanson 1980]. Allocators using this layout include Hoard [Berger et al. 2000a] and the Boehm-Demers-Weiser collector [Boehm and Weiser 1988].

4.4.1 *The Static Allocator.* Unlike the adaptive allocator, which allocates from increasingly large miniheaps, the static allocator allocates from a large, statically-sized heap whose size is set at compile time (see Figure 3). The heap is evenly divided into 12 regions (one for each power-of-two size class from 8 bytes to 16 kilobytes), so the heap must be at least $12M$ as large as the maximum amount of memory needed in any size class. While the heap itself is lazily allocated so that unused partitions do not consume physical memory, its address space requirements limit its use on 32-bit platforms to applications whose footprint is no more than 90MB in any size class ($2^{31}/24$).

Each heap region has an associated but separate bitmap, where one bit stands for one

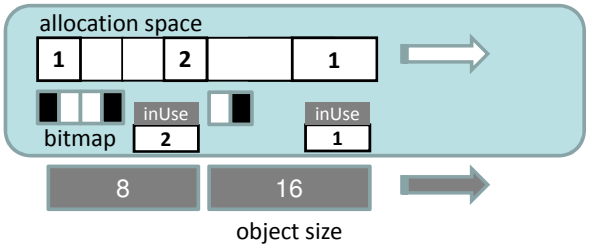


Fig. 3. The static (original) DieHard heap layout. A single statically-sized heap is divided into separate regions for each size class, within which objects are laid out randomly.

object. All bits are initially zero, indicating that the objects are all free. Allocation proceeds by rounding up the size request to the nearest power of two, and repeatedly choosing a random bit for the appropriate size class until it finds a zero bit. As Section 4.2 shows, this operation takes expected constant-time. Deallocation locates the appropriate bit by bit-masking, and then resets it, making the cost of deallocation $O(1)$ in the static variant (versus expected $O(1)$ for the adaptive algorithm).

4.5 Discussion

The design of DieHard's allocation algorithm departs significantly from previous memory allocators. In particular, it makes no effort to improve locality and can increase space consumption.

Locality: Many allocators attempt to increase spatial locality by placing objects that are allocated at the same time near each other in memory [Chilimbi et al. 1999; Feng and Berger 2005; Lea 1997; Wilson et al. 1995]. DieHard's random allocation algorithm instead makes it likely that such objects will be distant. This spreading out of objects has little impact on L1 locality because typical heap objects are near or larger than the L1 cache line size (32 bytes on the x86). However, randomized allocation leads to a large number of TLB misses in one application (see Section 7.2), and leads to higher resident set sizes because it can induce poor page-level locality. To maintain performance, the in-use portions of the DieHard heap should fit into physical RAM.

Space consumption: DieHard generally consumes more memory than conventional memory allocators. This increase in memory is caused by two factors: rounding up objects to the next power of two, and requiring that the heap be M times larger than necessary, although this is a tuneable parameter.

The rounding up of objects to the next power of two can, in the worst-case, increase memory consumption by up to a factor of two. Wilson et al. present empirical results suggesting that this policy can lead to significant fragmentation [Wilson et al. 1995]. Nonetheless, such an allocator is used in real systems, including FreeBSD's PHKmalloc [Kamp], and is often both time and space-efficient in practice [Feng and Berger 2005].

Any increase in memory consumption caused by rounding is balanced by two features of DieHard that reduce memory consumption. First, unlike conventional allocators like the GNU libc allocator, DieHard's allocator has no per-object headers. These headers typically consume eight bytes, but DieHard's per-object overhead is just one bit in the al-

location bitmap. Second, although coarse size classes can increase internal fragmentation, DieHard's use of segregated regions eliminates external fragmentation. The Lea allocator's external fragmentation plus per-object overhead increases memory consumption by approximately 20% [Feng and Berger 2005].

A more serious concern is the requirement of a factor of M additional space, multiplied by the number of replicas. We note that approaches like conservative garbage collection can impose an additional space overhead of 3X-5X over `malloc`/`free` [Hertz and Berger 2005; Zorn 1993]. DieHard also increases physical memory requirements and reduces available address space. These effects may make DieHard unsuitable for applications with large heap footprints running on 32-bit systems. We expect the problem of reduced address space will become less of an issue as 64-bit processors become commonplace. We also believe that DieHard's space-reliability tradeoff will be acceptable for many purposes, especially long-running applications with modest-sized heaps.

5. REPLICATION

While replacing an application's allocator with DieHard reduces the likelihood of memory errors, this stand-alone approach cannot detect uninitialized reads. To catch these errors, and to further increase the likelihood of correct execution, we have built a version of DieHard (currently for UNIX platforms only) that executes several replicas simultaneously. Figure 1 depicts the architecture, instantiated with three replicas.

The `diehard` command takes three arguments: the path to the replicated variant of the DieHard memory allocator (a dynamically-loadable library), the number of replicas to create, and the application name.

5.1 Replicas and Input

DieHard forks off replicas as separate processes, each with the LD_PRELOAD environment variable pointing to the DieHard memory management library `libdiehard_r.so`. This *library interposition* redirects all calls to `malloc` and `free` in the application to DieHard's memory manager. Because the memory manager picks a different random number generation seed on every invocation, all replicas execute with different sequences of random numbers.

DieHard uses both pipes and shared memory to communicate with the replicas. Each replica receives its standard input from DieHard via a pipe. Each replica then writes its standard output into a memory-mapped region shared between DieHard and the replica. After all I/O redirection is established, each replica begins execution, receiving copies of standard input from the main DieHard process.

While the stand-alone version of DieHard works for any program, the replicated DieHard architecture is intended for programs whose output is largely deterministic. The current implementation is targeted at standard UNIX-style commands that read from standard input and write to standard output. Also, while we intend to support programs that modify the filesystem or perform network I/O, these are not supported by the current version of the replicated system. We leave the use of DieHard replication with interactive applications as future work.

5.2 Voting

DieHard manages output from the replicas by periodically synchronizing at barriers. Whenever all currently-live replicas terminate or fill their output buffers (currently 4K each, a

pipe's unit of transfer), the *voter* compares the contents of each replica's output buffer. If all agree, then the contents of one of the buffers are sent to standard output, and execution proceeds as normal.

However, if not all of the buffers agree, it means that at least one of the replicas has an error. The voter then chooses an output buffer agreed upon by at least two replicas and sends that to standard out. Two replicas suffice, because the odds are slim that two randomized replicas with memory errors would return the same result.

Any non-agreeing replicas have either exited abnormally before filling their output buffers, or produced different output. Whenever a replica crashes, DieHard receives a signal and decrements the number of currently-live replicas. A replica that has generated anomalous output is no longer useful since it has entered into an undefined state. Our current implementation kills such failed replicas and decreases the currently-live replica count. To further improve availability, we could replace failed replicas with a copy of one of the "good" replicas with its random seed set to a different value.

5.3 Discussion

Executing applications simultaneously on the same system while both providing reasonable performance and preserving application semantics is a challenge. We address these issues here.

In order to make correct replicas output-equivalent to the extent possible, we intercept certain system calls that could produce different results. In particular, we redirect functions that access the date and system clock so that all replicas return the same value. We discuss a generalization of this approach in Section 9.

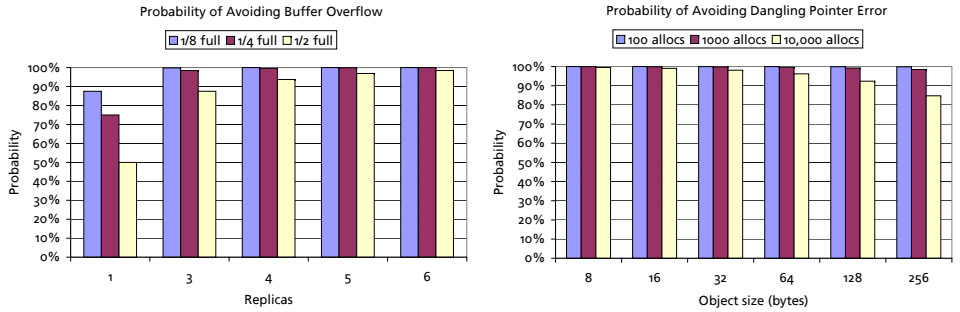
While it may appear that voting on all output might be expensive, it is amortized because this processing occurs in 4K chunks. More importantly, voting is only triggered by I/O, which is already expensive, and does not interfere with computation.

One disadvantage of the barrier synchronization employed here is that an erroneous replica could theoretically enter an infinite loop, which would cause the entire program to hang because barrier synchronization would never occur. There are two approaches that one can take: use a timer to kill replicas that take too long to arrive at the barrier, or ignore the problem, as we currently do. Establishing an appropriate waiting time would solve the problem of consensus in the presence of Byzantine failures, which is undecidable [Fischer et al. 1985].

6. ANALYSIS

While DieHard is immune to heap corruption caused by double frees, invalid frees, and heap metadata overwrites caused by overflow, it is probabilistically resilient to other memory errors. In this section, we quantify the probabilistic memory safety provided by both the stand-alone and replicated versions of DieHard. We derive equations that provide lower bounds on the likelihood of avoiding buffer overflow and dangling pointer errors, and detecting uninitialized reads. We assume that heap metadata, which is placed randomly in memory and protected on either side by guard pages, is not corrupted.

We use the following notation throughout the analyses. Recall that M denotes the heap expansion factor that determines how large the heap is relative to the maximum application live object size. We use k for the number of replicas, H for the maximum heap size, L for the maximum live size ($L \leq H/M$), and F for the remaining free space ($H - L$). When analyzing buffer overflows, we use O to stand for the number of objects' worth of bytes



(a) Probability of tolerating single-object buffer overflows for varying replicas and degrees of heap fullness.

(b) Probability of tolerating dangling pointer errors *using the stand-alone version of DieHard*, for varying object sizes and intervening allocations.

Fig. 4. Probabilities of tolerating buffer overflows and dangling pointer errors.

overflowed (e.g., a nine byte overflow could overwrite $O = 2$ eight byte objects). For dangling pointer errors, we use A to denote the number of allocations that have taken place after a premature call to `free`.

We also assume conservatively that all object requests are for a specific size class. This approach is conservative because the separation of different size classes improves the odds of avoiding memory errors. We also assume that there is either one replica or at least three, since the voter cannot decide which of two disagreeing replicas is the correct one.

Note that the analyses below quantify the probability of avoiding a *single* error of a given type. One can calculate the probability of avoiding multiple errors by multiplying the probabilities of avoiding each error, although this computation depends on an assumption of independence that may not hold. Also, these results only hold for objects smaller than 16K in size, because larger objects are managed separately as described in Section 4.1.

6.1 Tolerating Dangling Pointers

A dangling pointer error occurs when an object is freed prematurely, its address is reused for a subsequent allocation, and its contents are overwritten when it is returned by a subsequent call to `malloc`. Suppose that the object should have been freed A allocations later than it was; that is, the call to `free` should have happened at some point after the next A calls to `malloc` but before the $A+1$th call. Avoiding a dangling pointer error is thus the likelihood that some replica has not overwritten the object's contents after A allocations:

THEOREM 1. *Let Overwrites be the number of times that a particular freed object of size S gets overwritten by one of the next A allocations. Then the probability of this object being intact after A allocations, assuming $A \leq F/S$ and $k \neq 2$, is:*

$$P(\textit{Overwrites} = 0) \;\geq\; 1 - \left(\frac{A}{F/S}\right)^{k}.$$

PROOF. The prematurely freed object is indexed by one of the $Q = F/S$ bits in the allocation bitmap for its size class. The odds of a new allocation not overwriting that object

are thus $(Q-1)/Q$. Assume that after each allocation, we do not free an object, which is the worst case. After the second allocation, the odds are $(Q-1)/Q * (Q-2)/(Q-1) = (Q-2)/Q$. In general, after A allocations, the probability of not having overwritten a particular slot is $(Q-A)/Q$.

The probability that no replica has overwritten a particular object after A allocations is then one minus the odds of all of the replicas overwriting that object, or $1-(1-(Q-A)/Q)^k = 1-(A/(F/S))^k$. □

This result shows that DieHard is robust against dangling pointer errors, especially for small objects. For example, when $M = 2$ and 16MB 8-byte objects have been allocated, the stand-alone version of DieHard has a 99% chance of tolerating an 8-byte object freed 10,000 allocations too soon. Figure 4(b) graphically presents the probabilities of avoiding dangling pointer errors for different object sizes and numbers of intervening allocations.

6.2 Tolerating Buffer Overflows

We next derive the probability of tolerating buffer overflows. A buffer overflow can be tolerated if the following conditions hold in at least one replica: (1) the overflow does not overwrite any live data, and (2) the overflowed data is not overwritten by a subsequent allocation.

While buffer overflows are generally writes just beyond an allocated object, for our analysis, we model a buffer overflow as a write to any location in the heap. The following formula gives the probability that an overwrite satisfies condition (1), i.e., that it does not overwrite any live data.

THEOREM 2. *Let OverflowedObjects be the number of live objects overwritten by a buffer overflow. Then for $k \neq 2$, the probability of tolerating a buffer overflow is*

$$P(\mathit{OverflowedObjects} = 0) \;=\; 1 - \left[1 - \left(\tfrac{F}{H}\right)^{O}\right]^{k}.$$

PROOF. The odds of O objects overwriting at least one live object are 1 minus the odds of them overwriting no live objects, or $1-(\frac{F}{H})^O$. Tolerating the buffer overflow requires that at least one of the k replicas not overwrite any live objects, which is the same as 1 minus all of them overwriting at least one live object = $1-(1-(\frac{F}{H})^O)^k$. □

In addition to not overwriting any live data, tolerating a buffer overflow requires that condition (2) also hold: its contents must not be overwritten by a subsequent allocation before the overflowed data dies. This probability is the same as that of tolerating O dangling pointer errors.

Because dangling pointer protection is so effective, the likelihood of tolerating a buffer overflow is determined primarily by the live fraction of the heap. For example, when the heap is no more than $1/8$ full, DieHard in stand-alone mode provides an 87.5% chance of tolerating a single-object overflow, while three replicas avoids such errors with greater than 99% probability. Figure 4(a) shows the probability of protecting against overflows for different numbers of replicas and degrees of heap fullness.

6.3 Detecting uninitialized reads

We say that DieHard detects an uninitialized read when the read causes all of the replicas to differ on their output, leading to termination. An uninitialized read is a use of memory

obtained from an allocation before it has been initialized. If an application relies on values read from this memory, then its behavior will eventually reflect this use. We assume that uninitialized memory reads are either benign or propagate to output.

The odds of detecting such a read thus depend both on how much use the application makes of the uninitialized memory, and its resulting impact on the output. An application could **widen** the uninitialized data arbitarily, outputting the data in an infinite loop. On the other end of the spectrum, an application might **narrow** the data by outputting just one bit based on the contents of the entire uninitialized region. For example, it could output an 'A' if the first bit in the region was a 0, and 'a' if it was 1.

If we assume that the application generates just one bit of output based on every bit in the uninitialized area of memory, we get the following result:

THEOREM 3. *The probability of detecting an uninitialized read of B bits in k replicas ($k > 2$) in a* non-narrowing, non-widening *computation is:*

$$P(\textit{Detect uninitialized read}) = \frac{2^B!}{(2^B-k)!2^{Bk}}.$$

PROOF. For DieHard to detect an uninitialized read, all replicas must disagree on the result stemming from the read. In other words, all replicas must have filled in the uninitialized region of length B with a different B-bit number. There are 2^B numbers of length B, and k replicas yields 2^{Bk} possible combinations of these numbers. There are $(2^B)!/(2^B-k)!$ ways of selecting different B-bit numbers across the replicas (assuming $2^B > k$). We thus have a likelihood of detecting an uninitialized read of $(2^B!)/(2^B-k)!2^{Bk}$. □

Interestingly, in this case, replicas lower the likelihood of memory safety. For example, the probability of detecting an uninitialized read of four bits across three replicas is 82%, while for four replicas, it drops to 66.7%. However, this drop has little practical impact for reads of more data. The odds of detecting an uninitialized read of 16 bits drops from 99.995% for three replicas to 99.99% for four replicas.

DieHard's effectiveness at finding uninitialized reads makes it useful as an error-detecting tool during development. During experiments for this article, we discovered uninitialized reads in several benchmarks. The replicated version of DieHard typically terminated in several seconds. We verified these uninitialized read errors with Valgrind, which ran approximately two orders of magnitude slower.

7. EXPERIMENTAL RESULTS

We first measure the runtime impact of the DieHard memory manager on a suite of benchmark applications. We then empirically evaluate its effectiveness at avoiding both injected faults and actual bugs.

7.1 Benchmarks

We evaluate DieHard's performance with both the full SPECint2000 suite [Standard Performance Evaluation Corporation] running reference workloads, as well as a suite of allocation-intensive benchmarks. These benchmarks perform between 100,000 and 1.7 million memory operations per second (see Berger, Zorn and McKinley [Berger et al. 2001] for a detailed description). We include these benchmarks both because they are

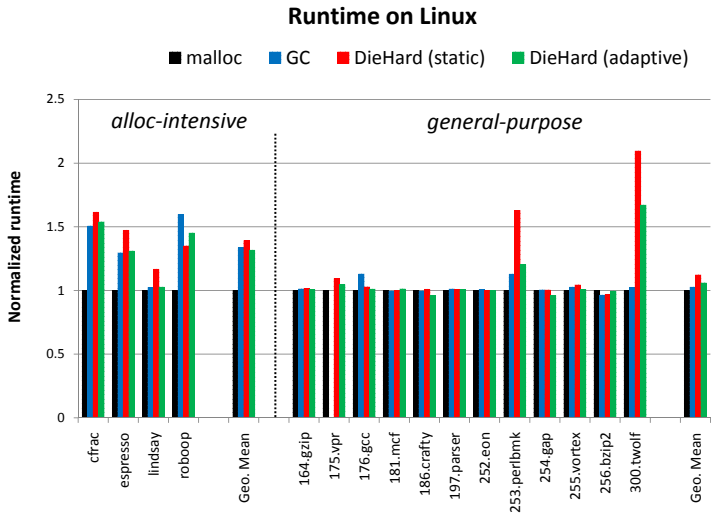


Fig. 5. Performance of the default `malloc`, the original version of DieHard, and the new adaptive version (all executing stand-alone), across a range of allocation-intensive and general-purpose benchmark applications.

widely used in memory management studies [Berger et al. 2000b; Grunwald et al. 1993; Johnstone and Wilson 1997] and because their unusually high allocation-intensity stresses memory management performance.

We run our benchmarks on two different platforms: Linux and Solaris. The Linux platform is a dual-processor Intel Xeon system with each 3.06GHz processor (hyperthreading active) equipped with 512K L2 caches and with 3 gigabytes of RAM. All code on Linux is compiled with g++ version 4.0.2. The Solaris platform is a Sun SunFire 6800 server, with 16 900MHz UltraSparc v9 processors and 16 gigabytes of RAM; code there is compiled with g++ 3.2. All code is compiled at the highest optimization level on all platforms. Timings are performed while the systems are quiescent. We report the average of three runs. Observed variances are below 1% for all experiments except for one, 300.twolf with DieHard, where the variance is 12%.

7.2 Overhead

We compare the adaptive and static versions of DieHard to both the Boehm-Demers-Weiser collector (version 6.8) and the default GNU libc allocator (glibc version 2.3.4), a variant of the Lea allocator [Lea 1997]. We run these experiments on our Linux platform. For the adaptive DieHard, the heap multiplication factor M is set to two unless stated otherwise; the static version uses a fixed heap size of 384 megabytes (32 megabytes per size class), the minimum required to run all of the benchmarks with the same heap configuration. The Boehm-Demers-Weiser collector is included for comparison because it represents an alternative trade-off in the design space between space, execution time, and safety guarantees. In particular, the BDW collector prevents dangling pointer errors, while DieHard probabilistically avoids them. However, the BDW collector provides no protection against buffer overflows.

For all experiments, we disable the replacement of unsafe library functions (see Section 4.4) for these experiments to isolate the protection that randomization and replication

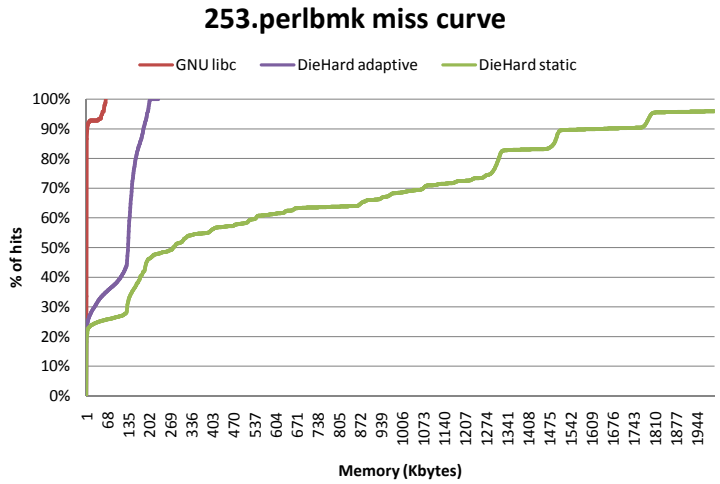


Fig. 6. Miss curve for 253.perlbmk with the default `malloc`, the original version of DieHard, and the new adaptive version (all executing stand-alone).

provide.

Runtime overhead

Figure 5 presents runtime overhead results. For the allocation-intensive benchmarks, DieHard suffers a performance penalty ranging from 2.5% to 53.8% (geometric mean: 31.9%), improving over the original (from 16.5% to 61.3% (geometric mean: 39%). Its overhead is somewhat lower than that suffered by the Boehm-Demers-Weiser collector (2.4% to 59.7%, geometric mean 33.7%).

However, DieHard's runtime overhead is substantially lower for most of the SPECint2000 benchmarks. The geometric mean of DieHard's overhead is 6%. As before, adaptive DieHard outperforms the static version, whose geometric mean overhead is 12%. DieHard performs somewhat worse than the Boehm-Demers-Weiser collector (geometric mean 2.5%). This average excludes the 175.vpr benchmark, because the Boehm-Demers-Weiser collector fails with a segmentation fault.

For two applications, DieHard degrades performance substantially, although adaptive DieHard reduces this overhead over the static version: 253.perlbmk (48.8% static, 20% adaptive) and 300.twolf (109% static, 67% adaptive). The 253.perlbmk benchmark is allocation-intensive, spending around 12.5% of its execution doing memory operations. This allocation intensity is responsible for both DieHard's and Boehm-Demers-Weiser's runtime overhead (12.7%). However, 300.twolf's overhead is due not to the cost of allocation but to TLB misses. 300.twolf uses a wide range of object sizes. In DieHard, accesses to these objects are spread over separate memory regions. The adaptive variant yields fewer TLB misses because it grows memory adaptively, while the static version spreads all objects over a larger heap.

Space overhead: To further explore the effect of the variants of DieHard on page-level locality, we gather page-level references using a tool that intercepts system memory

allocation calls (brk, sbrk, mmap, etc.) to track heap pages, and traps memory references by page protection. We use the SAD (Safely-Allowed-Drop) algorithm to reduce the trace to a manageable size [Kaplan et al. 2003]. We then run these traces through an LRU simulator to generate page miss curves that indicate the number of misses (page faults) that would arise for every possible size of available memory.

Figure 6 shows the miss curves for one of the outlier benchmarks, 253.perlbmk. Each point on the graph corresponds to the amount of memory (on the x-axis) required to hold a given percentage of page references (on the y-axis). The GNU libc allocator is effective at maintaining a small footprint, rising steeply until it hits a plateau, after which it rises again to reach its full footprint. The large memory space required by the static version of DieHard, combined with its random access pattern, gives it a low curve with a long tail. This kind of curve interacts poorly with virtual memory systems, because additional memory provides little benefit. However, by controlling its memory consumption, the current version of DieHard exhibits a miss curve closer in shape to that of the libc allocator.

7.2.1 *Solaris: Replicated Experiments.* To quantify the overhead of the replicated framework and verify its scalability, we measure running time with sixteen replicas on a 16-way Sun server. We ran these experiments with the allocation-intensive benchmark suite, except for lindsay, which has an uninitialized read error that DieHard detects and terminates. Running 16 replicas simultaneously increases runtime by approximately 50% versus running a single replica with the replicated version of the runtime (`libdiehard_r.so`). Part of this cost is due to process creation, which longer-running benchmarks would amortize. This result shows that while voting and interprocess communication impose some overhead, the replicated framework scales to a large number of processors.

7.3 Error Avoidance

We evaluate DieHard's effectiveness at avoiding both artificially-injected bugs and actual bugs in two real applications, the Squid web caching server and the Mozilla Firefox web browser.

7.3.1 *Fault Injection.* We implement two libraries that inject memory errors into unaltered applications running on UNIX platforms to explore the resilience of different runtime systems to memory errors including buffer overflows and dangling pointers.

We first run the application with a tracing allocator that generates an allocation log. Whenever an object is freed, the library outputs a pair, indicating when the object was allocated and when it was freed (in allocation time). We then sort the log by allocation time and use a fault-injection library that sits between the application and the memory allocator. The fault injector triggers errors probabilistically, based on the requested frequencies. To trigger an overflow, it requests less memory from the underlying allocator than was requested by the application. To trigger a dangling pointer error, it uses the log to invoke `free` on an object before it is actually freed by the application, and ignores the subsequent (actual) call to `free` this object. The fault injector only inserts dangling pointer errors for small object requests (< 16K).

We verified DieHard's resilience by injecting errors in both the espresso and cfrac benchmarks, running each one hundred times with the default allocator and with DieHard. The large number of allocations in both benchmarks make them particularly vulnerable to memory errors: espresso allocates approximately 4.5 million objects, while cfrac allocates almost 11 million objects [Berger et al. 2001].

Dangling pointer errors: We first introduce dangling pointers of frequency of 0.5% with distance 10: one out of every two hundred objects is freed ten allocations too early. This high error rate prevents both espresso and cfrac from running to completion with the default allocator in any run. With DieHard, espresso runs correctly in 81 of the 100 runs, while cfrac runs successfully 36% of the time.

Buffer overflows: We then injected buffer overflow errors at a 1% rate (1 out of every 100 allocations), under-allocating object requests of 32 bytes or more by 4 bytes. With the default allocator, espresso crashes or enters an infinite loop in every run. With DieHard, it runs successfully 66% of the time. Similarly, cfrac also fails to complete successfully in every attempt with the default allocator, but with DieHard, it runs successfully 97% of the time.

Discussion: The resilience of these benchmarks to injected faults is substantially higher than what one would expect given the analytical lower bounds from Section 6. Both experiments inject hundreds of thousands of dynamic errors. In particular, the analytical lower bounds for correct execution in the face of this number of buffer overflows errors is near zero. These results suggest that certain aspects of the analysis, especially the modeling of overflows as writes to arbitrary areas of the heap, may be too conservative. Internal characteristics of each benchmark also play a difficult to quantify but important role in making them resilient to faults.

7.3.2 *Real Faults.* We also tested DieHard on two actual buggy applications. Version 2.3s5 of the Squid web cache server has a buffer overflow error that can be triggered by an ill-formed input. When faced with this input and running with either the GNU libc allocator or the Boehm-Demers-Weiser collector, Squid crashes with a segmentation fault. Using DieHard in stand-alone mode, the overflow has no effect.

We also used DieHard with several versions of the Mozilla web browser. While the Mozilla suite is a mature and extensively tested program, heap-based security vulnerabilities continue to be discovered. We test vulnerable versions of Mozilla or Firefox with an HTML file that contained two separate overflow errors: with version 1.0.2, bug number 307259 (an IDN heap overflow) and with version 1.7.2, bug number 251381 (an overflow in PNG image processing). [2] We perform ten experiments loading this HTML file, and then attempting to load a complex web page from a popular news site.

Without DieHard, both variants of Mozilla fail to run successfully in all ten attempts. Because Mozilla is a multithreaded program, the allocation sequence varies from run to run, so the effect of the overflows varies: in some cases, Mozilla crashes immediately, while in others, it crashes when entering a URL. DieHard prevents Mozilla 1.0.2 from crashing 7 out of 10 runs, and protects Mozilla 1.7.2 60% of the time. However, in the latter case, DieHard's protection is not entirely complete: in three of the successes, while program execution continues successfully, an error window appears when closing Mozilla.

Public deployment experience: To increase the potential user base of DieHard, we developed a version for the Windows operating system. Unlike UNIX-based operating systems, Windows makes it relatively difficult to replace memory allocation functions in

[2]Detailed information on these bugs, including proofs-of-concept, is available at https://bugzilla.mozilla.org/show_bug.cgi?id=⟨bugnumber⟩.

Error	glibc	BDW GC	CCured	Rx	Failure-oblivious	DieHard
heap corruption	⊥	⊥	`abort`	✓	⊥	✓
invalid frees	⊥	✓	✓	⊥	⊥	✓
double frees	⊥	✓	✓	✓	⊥	✓
dangling pointers	⊥	✓	✓	⊥	⊥	✓*
buffer overflows	⊥	⊥	`abort`	⊥	⊥	✓*
uninitialized reads	⊥	⊥	`abort`	⊥	⊥	`abort`*

Table I. This table compares how various systems handle memory safety errors: ✓denotes correct execution, ⊥ denotes an undefined result, and `abort` means the program terminates abnormally. See Section 8 for a detailed explanation of each system. The DieHard results for the last three errors (marked with asterisks) are probabilistic; see Section 6 for exact formulae.

compiled Windows executables. We used a publicly-available "hooking" tool called MadCodeHook that enables this functionality [Mathias Rauen], and targeted it specifically for the Mozilla Firefox web browser.

We chose Firefox as our target first because it is a widely-used web browser, and browsers are especially at risk if they have security vulnerabilities. As described above, DieHard protects Firefox from the effects of several previous vulnerabilities. Second, the memory footprint of a browser is modest with respect to usual memory configurations (on the order of 50MB). In our tests, DieHard roughly doubled memory consumption (in one scenario, Firefox consumed 5198 total pages without Mozilla, and 11327 total pages with DieHard), making it likely to remain in physical RAM for most users. Finally, unlike other widely-used applications (notably those from Microsoft), which statically link to their memory management libraries or employ custom memory allocators, Firefox makes extensive use of the memory allocator directly via the standard `malloc` interface.

We publicly announced this version of DieHard at the end of December 2006. Shortly afterward, a news story covering DieHard was picked up by various press outlets, resulting in a large number of visits to the web site. Many of these visitors subsequently downloaded the software (over 10,500 downloads during January 2006). We view this as anecdotal evidence that, for many, the additional reliability and security that DieHard provides outweighs its impact on memory consumption. User reports have been positive: the two most requested features are coverage of more applications, and the ability to have DieHard run all the time (it currently requires activation after every re-boot).

8. RELATED WORK

This section describes related work in software engineering, fault tolerance, memory management, approaches to address security vulnerabilities, fail-stop, debugging and testing. Table I summarizes how DieHard and the systems described here handle memory safety errors.

Our approach is inspired by **N-version programming**, in which independent programmers produce variants of a desired program [Avizienis 1985]. Whereas N-version programming relies on a conjecture of independence across programmers to reduce the likelihood of errors, DieHard provides hard analytical guarantees.

Fault tolerance: DieHard's use of replicas with a voter process is closely related to Bressoud and Schneider's hypervisor-based system, which provides fault tolerance in the

face of fail-stop executions [Bressoud and Schneider 1995]. In addition to supporting replication and voting, their hypervisor eliminates all non-determinism. This approach requires hardware support or code rewriting, while DieHard's voter is less general but lighter weight.

Rinard et al. present a compiler-based approach called *boundless buffers* that caches out-of-bound writes in a hash table for later reuse [Rinard et al. 2004]. This approach eliminates buffer overflow errors (though not dangling pointer errors), but requires source code and imposes higher performance overheads (1.05x to 8.9x).

Several researchers have proposed unsound techniques that can prevent programs from crashing [Dhurjati et al. 2003; Qin et al. 2005; Rinard et al. 2004]. *Automatic pool allocation* segregates objects into pools of the same type, thus ensuring that dangling pointers are always overwritten only by objects of the same type [Dhurjati et al. 2003]. While this approach yields type safety, the resulting program behavior is unpredictable. *Failure-oblivious systems* continue running programs by ignoring illegal writes and manufacturing values for reads of uninitialized areas [Rinard et al. 2004]. These actions impose as high as 8X performance overhead and can lead to incorrect program execution. Rx uses checkpointing and logging in conjunction with a versioning file system to recover from *detectable* errors, such as crashes. After a crash, Rx rolls back the application and restarts with an allocator that selectively ignores double frees, zero-fills buffers, pads object requests, and defers frees [Qin et al. 2005]. Because Rx relies on checkpointing and rollback-based recovery, it is not suitable for applications whose effects cannot be rolled back. It is also unsound: Rx cannot detect *latent* errors that lead to incorrect program execution rather than crashes.

Memory management approaches: Typical runtime systems sacrific robustness in favor of providing fast allocation with low fragmentation. Most implementations of `malloc` are susceptible to both double frees and heap corruption caused by buffer overflows. However, some recent memory managers detect heap corruption, including version 2.8 of the Lea allocator [Lea 1997; Robertson et al. 2003], while others (Rockall [Ball et al. 2001], dnmalloc [Younan et al. 2005], Heap Server [Kharbutli et al. 2006]) fully segregate metadata from the heap like DieHard, preventing heap corruption.

Garbage collection avoids dangling pointer errors but requires a significant amount of space to provide reasonable performance (3X-5X more than `malloc`/`free`) [Hertz and Berger 2005; Swamy et al. 2006; Zorn 1993]. DieHard ignores double and invalid frees and segregates metadata from the heap to avoid overwrites, but unlike the Boehm-Demers-Weiser collector, its avoidance of dangling pointers is probabilistic rather than absolute. Unlike previous memory managers, DieHard provides protection of heap data (not just metadata) from buffer overflows, and can detect uninitialized reads.

Security vulnerabilities: Previous efforts to reduce vulnerability to heap-based security attacks randomize the base address of the heap [Bhatkar et al. 2003; PaX Team] or randomly pad allocation requests [Bhatkar et al. 2005]. Base address randomization provides little protection from heap-based attacks on 32-bit platforms [Shacham et al. 2004]. Although protection from security vulnerabilities is not its intended goal, DieHard makes it difficult for an attacker to predict the layout or adjacency of objects in any replica.

Fail-stop approaches: A number of approaches that attempt to provide type and memory safety for C (or C-like) programs are fail-stop, aborting program execution upon detecting an error [Austin et al. 1994; Avots et al. 2005; Necula et al. 2002; Xu et al. 2004; Yong and Horwitz 2003]. We discuss two representative examples: Cyclone and CCured.

Cyclone augments C with an advanced type system that allows programmers direct but safe control over memory [Jim et al. 2002]. CCured instruments code with runtime checks that dynamically ensure memory safety and uses static analysis to remove checks from places where memory errors cannot occur [Necula et al. 2002]. While Cyclone uses region-based memory management and safe explicit deallocation [Grossman et al. 2002; Swamy et al. 2006], CCured relies on the BDW garbage collector to protect against double frees and dangling pointers. Unlike DieHard, which works with binaries and supports any language using explicit allocation, both Cyclone and CCured operate on an extended version of C source code that typically requires manual programmer intervention. Both abort program execution when detecting buffer overflows or other errors, while DieHard can often avoid them.

Debugging and testing: Tools like Purify [Hastings and Joyce 1991] and Valgrind [Nethercote and Seward 2007] (with the Memcheck tool [Seward and Nethercote 2005; Nethercote and Fitzhardinge 2004]) use binary rewriting or emulation to dynamically detect memory errors in unaltered programs. However, these often impose prohibitive runtime overheads (2-25X) and space costs (around 10X) and are thus only suitable during testing. SWAT [Hauswirth and Chilimbi 2004] uses sampling to detect memory leaks at runtime with little overhead (around 5%), and could be employed in conjunction with DieHard.

9. CONCLUSION AND FUTURE WORK

DieHard is a runtime system that effectively tolerates memory errors and provides probabilistic memory safety. DieHard uses randomized allocation to give the application an approximation of an infinite-sized heap, and uses replication to further increase error tolerance and detect uninitialized memory reads that propagate to program output. DieHard allows an explicit trade-off between memory usage and error tolerance, and is useful for programs in which memory footprint is less important that reliability and security. We show that DieHard adds little CPU overhead to many of the SPECint2000 benchmark programs, while the CPU overhead in allocation-intensive programs is larger.

We show analytically that DieHard increases error tolerance, and confirm our analytic results by demonstrating that DieHard significantly increases the error tolerance of an application in which faults are artifically injected. We also describe experiments in which DieHard successfully tolerates known buffer-overflow and dangling pointer errors in the Squid web cache server and the Mozilla Firefox web browser.

The DieHard runtime system tolerates heap errors but does not prevent safety errors based on stack corruption. We believe that with compiler support, the ideas proven successful in DieHard could be used to improve error tolerance on the stack and also in object field references. We plan to investigate the effectiveness of this approach in future work.

One limitation of the replicated form of DieHard is its inability to work with programs that generate non-deterministic output or output related to environmental factors (e.g., time-of-day, performance counters, interactive events, etc.) We are developing a "Determinator" that enforces the following invariant: the n^{th} system call returns the same value across all replicas. The Determinator consists of two parts: a Determinator service and a set of replacement system calls that communicate with the Determinator service. When a DieHard replica executes a system call, it turns into a socket communication with the service. If this execution of the given system call has not yet been executed by any replica, the Determinator executes it on that replica's behalf. It communicates back the results to the replica, and caches them to provide in response to calls from the other replicas. We

also plan to incorporate mechanisms to ensure that non-determinism due to scheduling of multithreaded programs does not affect execution [Pool et al. 2007].

Improving the security and reliability of programs written in C and C++ is recognized by the research community as an important priority and many approaches have been suggested. In this article, we present a unique and effective approach to soundly tolerating memory errors in unsafe programs without requiring the programs be rewritten or even recompiled. Like garbage collection, DieHard represents a new and interesting alternative in the broad design space that trades off CPU performance, memory utilization, and program correctness.

ACKNOWLEDGMENTS

The authors would like to thank Mike Barnett, Mike Bond, Mark Corner, Trishul Chilimbi, Ted Hart, Mike Hicks, Daniel Jiménez, David Jensen, Scott Kaplan, Brian Levine, Andrew McCallum, David Notkin, and Gene Novark for their helpful comments. Thanks also to Shan Lu and Yuanyuan Zhou for providing us the buggy inputs for Squid.

DieHard is publicly available at http://www.diehard-software.org/.

REFERENCES

AUSTIN, T. M., BREACH, S. E., AND SOHI, G. S. 1994. Efficient detection of all pointer and array access errors. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 290–301.

AVIZIENIS, A. 1985. The N-version approach to fault-tolerant systems. *IEEE Transactions on Software Engineering 11,* 12 (Dec.), 1491–1501.

AVOTS, D., DALTON, M., LIVSHITS, V. B., AND LAM, M. S. 2005. Improving software security with a C pointer analysis. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*. ACM Press, New York, NY, USA, 332–341.

BALL, T., CHAKI, S., AND RAJAMANI, S. K. 2001. Parameterized verification of multithreaded software libraries. In *7th International Conference on Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science, vol. 2031. 158–173.

BERGER, E. D. 2006. Heapshield: Library-based heap overflow protection for free. Tech. Rep. UMCS TR-2006-28, Department of Computer Science, University of Massachusetts Amherst. May.

BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. 2000a. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*. Cambridge, MA, 117–128.

BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. 2000b. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA, 117–128.

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

BERGER, E. D., ZORN, B. G., AND MCKINLEY, K. S. 2001. Composing high-performance memory allocators. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001)*. Snowbird, Utah.

BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. 2003. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*. USENIX, 105–120.

BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*. USENIX, 271–286.

BOEHM, H.-J. AND WEISER, M. 1988. Garbage collection in an uncooperative environment. *Software Practice and Experience 18,* 9, 807–820.

BRESSOUD, T. C. AND SCHNEIDER, F. B. 1995. Hypervisor-based fault tolerance. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*. ACM Press, New York, NY, USA, 1–11.

CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. 1999. Cache-conscious structure layout. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*. ACM SIGPLAN Notices. ACM Press, Atlanta, 1–12.

DETLEFS, D. L. 1993. Empirical evidence for using garbage collection in C and C++ programs. In *OOPSLA/E-COOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, E. Moss, P. R. Wilson, and B. Zorn, Eds.

DHURJATI, D. AND ADVE, V. 2006. Backwards-compatible array bounds checking for c with very low overhead. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*. ACM Press, New York, NY, USA, 162–171.

DHURJATI, D., KOWSHIK, S., AND ADVE, V. 2006. Safecode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 144–157.

DHURJATI, D., KOWSHIK, S., ADVE, V., AND LATTNER, C. 2003. Memory safety without runtime checks or garbage collection. In *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*. ACM Press, San Diego, CA.

FENG, Y. AND BERGER, E. D. 2005. A locality-improving dynamic memory allocator. In *Proceedings of the ACM SIGPLAN 2005 Workshop on Memory System Performance (MSP)*. Chicago, IL.

FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM 32,* 2, 374–382.

GROSSMAN, D., MORRISETT, G., JIM, T., HICKS, M., WANG, Y., AND CHENEY, J. 2002. Region-based memory management in Cyclone. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 282–293.

GRUNWALD, D., ZORN, B., AND HENDERSON, R. 1993. Improving the cache locality of memory allocation. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*. ACM SIGPLAN Notices, vol. 28(6). ACM Press, Albuquerque, NM, 177–186.

HANSON, D. R. 1980. A portable storage management system for the Icon programming language. *Software Practice and Experience 10,* 6, 489–500.

HASTINGS, R. AND JOYCE, B. 1991. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter 1992 USENIX Conference*. San Francisco, California, 125–138.

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

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

JIM, T., MORRISETT, J. G., GROSSMAN, D., HICKS, M. W., CHENEY, J., AND WANG, Y. 2002. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 275–288.

JOHNSTONE, M. S. AND WILSON, P. R. 1997. The memory fragmentation problem: Solved? In *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, P. Dickman and P. R. Wilson, Eds.

KAEMPF, M. 2001. Vudo malloc tricks. *Phrack Magazine 57,* 8 (Aug.).

KAMP, P.-H. Malloc(3) revisited. http://phk.freebsd.dk/pubs/malloc.pdf.

KAPLAN, S. F., SMARAGDAKIS, Y., AND WILSON, P. R. 2003. Flexible reference trace reduction for VM simulations. *ACM Trans. Model. Comput. Simul. 13,* 1, 1–38.

KHARBUTLI, M., JIANG, X., SOLIHIN, Y., VENKATARAMANI, G., AND PRVULOVIC, M. 2006. Comprehensively and efficiently protecting the heap. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM Press, New York, NY, USA, 207–218.

LEA, D. 1997. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

MARSAGLIA, G. 1994. yet another RNG. posted to the electronic bulletin board sci.stat.math.

MATHIAS RAUEN. madCodeHook. http://www.madcodehook.com.

NECULA, G. C., MCPEAK, S., AND WEIMER, W. 2002. Ccured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 128–139.

NETHERCOTE, N. AND FITZHARDINGE, J. 2004. Bounds-checking entire programs without recompiling. In *SPACE 2004*. Venice, Italy.

NETHERCOTE, N. AND SEWARD, J. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 89–100.

PAX TEAM. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

POOL, J., SIN, I., AND LIE, D. 2007. Relaxed determinism: Making redundant execution on multiprocessors practical. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS 2007)*.

QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. 2005. Rx: Treating bugs as allergies: A safe method to survive software failures. In *Proceedings of the Twentieth Symposium on Operating Systems Principles*. Operating Systems Review, vol. XX. ACM, Brighton, UK.

RINARD, M., CADAR, C., DUMITRAN, D., ROY, D. M., AND LEU, T. 2004. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 2004 Annual Computer Security Applications Conference*.

RINARD, M., CADAR, C., DUMITRAN, D., ROY, D. M., LEU, T., AND WILLIAM S. BEEBEE, J. 2004. Enhancing server availability and security through failure-oblivious computing. In *Sixth Symposium on Operating Systems Design and Implementation*. USENIX, San Francisco, CA.

ROBERTSON, W., KRUEGEL, C., MUTZ, D., AND VALEUR, F. 2003. Run-time detection of heap-based overflows. In *LISA '03: Proceedings of the 17th Large Installation Systems Administration Conference*. USENIX, 51–60.

SEWARD, J. AND NETHERCOTE, N. 2005. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*. Anaheim, California, USA.

SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. 2004. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and Communications Security*. ACM Press, New York, NY, USA, 298–307.

STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC2000. http://www.spec.org.

SWAMY, N., HICKS, M., MORRISETT, G., GROSSMAN, D., AND JIM, T. 2006. Experience with safe manual memory management in cyclone. *Science of Computer Programming*. Special issue on memory management. Expands ISMM conference paper of the same name. To appear.

US-CERT. US-CERT vulnerability notes. http://www.kb.cert.org/vuls/.

WILSON, P. R., JOHNSTONE, M. S., NEELY, M., AND BOLES, D. 1995. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*. Lecture Notes in Computer Science, vol. 986. Springer-Verlag, Kinross, Scotland, 1–116.

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

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

YOUNAN, Y., JOOSEN, W., PIESSENS, F., AND DEN EYNDEN, H. V. 2005. Security of memory allocators for C and C++. Tech. Rep. CW 419, Department of Computer Science, Katholieke Universiteit Leuven, Belgium. July. Available at http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW419.pdf.

ZORN, B. 1993. The measured cost of conservative garbage collection. *Software Practice and Experience 23*, 733–756.