DieHard: Probabilistic Memory Safety for Unsafe Languages

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

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

Abstract

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 runtime system that tolerates these errors while probabilistically maintaining soundness. DieHard uses randomization and replication to achieve *probabilistic memory safety* by approximating an infinite-sized heap. DieHard's memory manager randomizes the location of objects in a heap that is at least twice as large as required. This algorithm prevents heap corruption and provides a probabilistic guarantee of avoiding memory errors. 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, the 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, including a heap-based buffer overflow in an actual application.

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

Keywords DieHard, probabilistic memory safety, randomization, replication, dynamic memory allocation

1. Introduction

While the use of safe languages is growing, many software applications are still 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 [39], are especially pernicious. These errors fall into the following categories:

Dangling pointers: If the 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 the contents of live objects on the heap.

Heap metadata overwrites: If heap metadata is stored near heap objects, an out-of-bound write can corrupt it.

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 [18] and Valgrind [28, 35] 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 [12, 20], 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 [1, 3, 27, 41, 42]. 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 — users are rarely happy to see their programs suddenly stop. Some systems instead sacrifice soundness in order to prolong execution in the face of memory errors [30, 32]. For example, failure-oblivious computing builds on a safe C compiler but drops illegal writes and manufactures values for invalid reads. Unfortunately, these systems provide no assurance to programmers that their programs are executing correctly.

This paper 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.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'06 June 10–16, 2006, Ottawa, Ontario, Canada
Copyright © 2006 ACM 1-59593-320-4/06/0006. . . $5.00.

randomized memory manager that allows the computation of the exact probabilities of detecting or avoiding memory errors.

The DieHard memory manager places objects randomly across a heap whose size is a multiple of the maximum required (Figure 1 shows an example heap layout). The resulting spacing between objects makes it likely that buffer overflows end up overwriting only empty space. Randomized allocation also makes it unlikely that a newly-freed object will soon be overwritten by a subsequent allocation, thus avoiding dangling pointer errors. It also improves application robustness by segregating all heap metadata from the heap (avoiding most heap metadata overwrites) and ignoring attempts to `free` already-freed or invalid objects. Despite its degradation of spatial locality, we show that the DieHard memory manager's impact on performance is small for many applications (average 8% across the SPECint2000 benchmark suite), and actually *improves* the performance of some applications when running on Windows XP.

While the stand-alone version of DieHard provides substantial protection against memory errors, the replicated version both increases the protection and detects errors caused by illegal reads. In this mode of operation, DieHard executes multiple replicas of the same program simultaneously, each with different seeds to their respective randomized allocators. Errors like buffer overflows are thus likely to overwrite different areas of memory in the different replicas. DieHard intercepts output from all of the various replicas and compares the contents of each before transmitting any output. With high probability, whenever any two programs agree on their output, they executed safely. In other words, in any agreeing replicas, any buffer overflows only overwrote dead data, and dangling pointers were never overwritten. If an application's output depends on uninitialized data, these data will be different across the replicas, and thus DieHard will detect them.

Since replacing the heap with DieHard significantly improves reliability, we believe that 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. Using additional replicas can further increase 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.

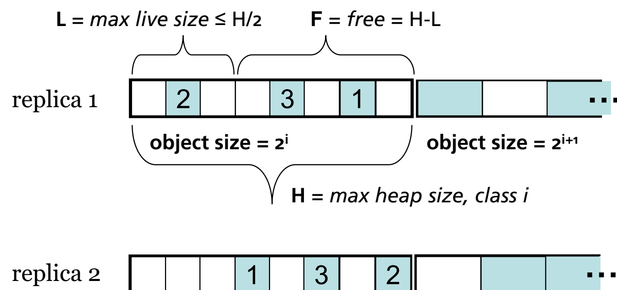


Figure 1. DieHard's heap layout. The heap is divided into separate regions, within which objects are laid out randomly. Notice the different layouts across replicas.

2.1 Outline

The rest of this paper is organized as follows. Section 3 formalizes the notions of probabilistic memory safety and 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.

3. Probabilistic Memory Safety

For the purposes of this paper, 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, we would ideally like an execution environment that would allow such programs to continue to execute correctly (soundly) in the face 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* [31]).

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 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. Finally, by randomizing the choice of freed objects to reclaim, recently-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. However, it does not quite capture infinite-heap semantics, because it does not detect uninitialized reads. In order to detect these, we require that the infinite heap and every allocated object be filled with random values. We can then detect uninitialized reads by simultaneously executing at least two *replicas* with different randomized allocators and comparing their

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

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 algorithm's initialization phase, and then describe the allocation and deallocation algorithms. For purposes of exposition, we refer to these as `DieHardMalloc` and `DieHardFree`, but in the actual implementation, these are simply called `malloc` and `free`. We use interposition to replace the calls in the target application; see Section 5.1 for details.

4.1 Initialization

The initialization step first obtains free memory from the system using `mmap`. The heap size is a parameter to the allocator, corresponding to the M factor described above. For the replicated version only, DieHard then uses its random number generator to fill 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`, a source of true randomness. The 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 [26].

The heap is logically partitioned into twelve regions, one for each power-of-two size class from 8 bytes to 16 kilobytes. Each region is allowed to become at most $1/M$ full. DieHard allocates larger objects 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 significantly speeds allocation by allowing expensive division and modulus operations to be replaced with bit-shifting. This organization also allows DieHard to efficiently prevent heap overflows caused by unsafe library functions like `strcpy`, as we describe in Section 4.4.

Separate regions are crucial to making the allocation algorithm practical. If instead objects were randomly spread across the entire heap area, significant 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.

Another vital aspect of the algorithm 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 [23]. 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 heap region, where one bit always stands for one object. All bits are initially zero, indicating that every object is free. Additionally, DieHard tracks the number of objects allocated to each region (`inUse`); this number is used to ensure that the number of objects does not exceed the threshold factor of $1/M$ in the partition.

4.2 Object Allocation

When an application requests memory from `DieHardMalloc`, the allocator first checks to see whether the request is for a large object (larger than 16K); if so, it uses `allocateLargeObject` to satisfy the request, which uses `mmap` and stores the address in a table for validity checking by `DieHardFree`. Otherwise, it converts the size request into a size class ($\lceil \log_2 \rceil$ of the request,

```
1  void DieHardInitHeap (int MaxHeapSize) {
2    // Initialize the random number generator
3    // with a truly random number.
4    rng.setSeed (realRandomSource);
5    // Clear counters and allocation bitmaps
6    // for each size class.
7    for (c = 0; c < NumClasses; c++) {
8      inUse[c] = 0;
9      isAllocated[c].clear();
10   }
11   // Get the heap memory.
12   heap = mmap (NULL, MaxHeapSize);
13   // REPLICATED: fill with random values
14   for (i = 0; i < MaxHeapSize; i += 4)
15     ((long *) heap)[i] = rng.next();
16 }
```

```
1  void * DieHardMalloc (size_t sz) {
2    if (sz > MaxObjectSize)
3      return allocateLargeObject(sz);
4    c = sizeClass (sz);
5    if (inUse[c] == PartitionSize / (M * sz))
6      // At threshold: no more memory.
7      return NULL;
8    // Probe for a free slot.
9    do {
10     index = rng.next() % bitmap size;
11     if (!isAllocated[c][index]) {
12       // Found one.
13       // Pick pointer corresponding to slot.
14       ptr = PartitionStart + index * sz;
15       // Mark it allocated.
16       inUse[c]++;
17       isAllocated[c][index] = true;
18       // REPLICATED: fill with random values.
19       for (i = 0; i < getSize(c); i += 4)
20         ((long *) ptr)[i] = rng.next();
21       return ptr;
22     }
23   } while (true);
24 }
```

```
1  void DieHardFree (void * ptr) {
2    if (ptr is not in the heap area)
3      freeLargeObject(ptr);
4    c = partition ptr is in;
5    index = slot corresponding to ptr;
6    // Free only if currently allocated;
7    if (offset correct &&
8        isAllocated[c][index]) {
9      // Mark it free.
10     inUse[c]--;
11     isAllocated[c][index] = false;
12   } // else, ignore
13 }
```

Figure 2. Pseudocode for DieHard heap initialization, object allocation and deallocation routines.

minus 3). As long as the corresponding region is not already full, it then looks for space.

Allocation then proceeds much like probing into a hash table. The allocator picks a random number and checks to see if the slot in the appropriate partition is available. The fact that the heap can only become $1/M$ full bounds the expected time to search for an unused slot to $\frac{1}{1-(1/M)}$. For example, for $M = 2$, the expected number of probes is two.

After finding an available slot, the allocator marks the object as allocated, increments the allocated count, and, for the replicated version, fills the object with randomized values. DieHard relies on this randomization to detect uninitialized reads, as we describe in Section 5.

4.3 Object Deallocation

To defend against erroneous programs, `DieHardFree` takes several steps to ensure that any object given to it is in fact valid. First, it checks to see if the address to be freed is inside the heap area, indicating it may be a large object. Because all large objects are `mmap`ed on demand, they lie outside of the main heap. The function `freeLargeObject` checks the table to ensure that this object was indeed returned by a previous call to `allocateLargeObject`. If so, it `munmap`s the object; otherwise, it ignores the request.

If the address is inside the small-object heap, DieHard checks it for validity to prevent double and invalid frees. First, the offset of the address from the start of its region (for the given size class) must be a multiple of the object size. Second, the object must be currently marked as allocated. If both of these conditions hold, DieHard finally resets the bit corresponding to the object location in the bitmap and decrements the count of allocated objects for this region.

4.4 Limiting Heap Buffer Overflows

While randomizing the heap provides probabilistic protection against heap buffer overflows (see Section 6.1), DieHard's heap layout 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 (two comparisons). If so, it 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 (two 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.

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 [11, 14, 25, 40]. 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 7.2.1), 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's memory management policies tend to consume 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.

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 [40]. Nonetheless, such an allocator is used in real systems, FreeBSD's PHKmalloc [24], and is both time and space-efficient in practice [14].

Any increase in memory consumption caused by rounding is balanced by two DieHard features that reduce memory consumption. First, unlike most conventional allocators including 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 allocation bitmap. Second, while coarse size classes can increase internal fragmentation, DieHard's use of segregated regions completely eliminates external fragmentation. The Lea allocator's external fragmentation plus per-object overhead increases memory consumption by approximately 20% [14].

A more serious concern is the requirement of a factor of M additional space for each of the twelve size classes, and the use of replicas. In the worst case, a program using DieHard could request objects of just one size and so require up to $12M$ more memory than needed. We could reduce this overhead using profile information to reserve only M times the maximum needed for each size class. However, Robson showed that this factor (a logarithm of the ratio of the largest size to the smallest) is the worst case for *all* memory allocators [34]. Approaches like conservative garbage collection can impose an additional space overhead of 3X-5X over `malloc/free` [20, 44]. Finally, memory that is reserved by DieHard but not used does not consume any virtual memory; the actual implementation of DieHard lazily initializes heap partitions. Nonetheless, DieHard's approach reduces the available address space, which may make it 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 3 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 spawns each replica in a separate process, each with the LD_PRELOAD environment variable pointing to the DieHard memory management library `libdiehard.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 inherently non-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, the unit of transfer of a pipe), 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.

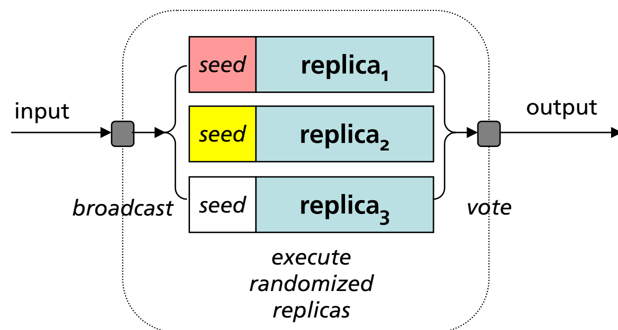


Figure 3. The replicated DieHard architecture. Input is broadcast to multiple replicas, each equipped with a different, fully-randomized memory manager. Output is only committed when at least two replicas agree on the result.

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 number generation 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.

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.

A 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 [15].

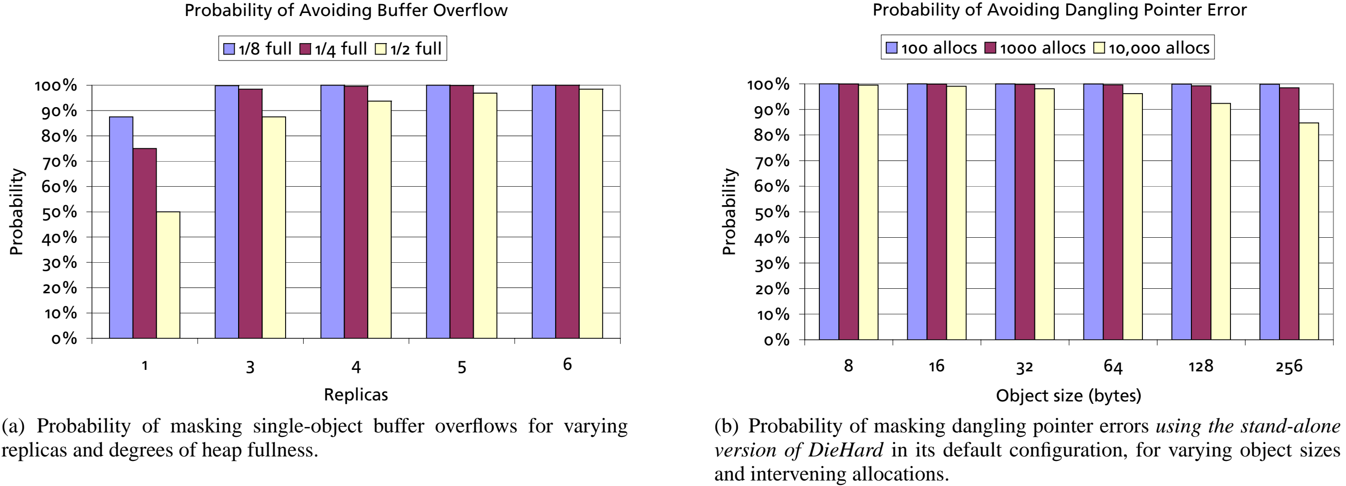
6. Analysis

While DieHard is immune to heap corruption caused by double frees, invalid frees, and heap metadata overwrites caused by overflow, its immunity to other memory errors is probabilistic. 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 the 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 the maximum heap size, L the maximum live size ($L \leq H/M$), and F the remaining free space ($H - L$). Figure 1 depicts these variables graphically. When analyzing buffer overflows, we use O to stand for the number of objects' worth of bytes overflowed (e.g., a 9-byte overflow could overwrite $O = 2$ 8-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 make a simplifying and conservative assumption in these analyses 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 as-



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

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

Figure 4. Probabilities of avoiding buffer overflows and dangling pointer errors.

sumption 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 Masking Buffer Overflows

In this section, we derive the probability of masking buffer overflows. 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. If a buffer overflow does not overwrite any live data in at least one replica's heap, we say that the buffer overflow has been successfully **masked**. The following formula gives the probability of successfully masking a buffer overflow.

Theorem 1. *Let OverflowedObjects be the number of live objects overwritten by a buffer overflow. Then for $k \neq 2$, the probability of masking a buffer overflow is*

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

Proof. The odds of O objects overwriting at least one live object are 1 minus the odds of them overwriting no live objects, or $1 - (\frac{F}{H})^{O}$. Masking 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 = $1 - (1 - (\frac{F}{H})^{O})^{k}$. □

Probabilistic memory safety provides good protection against modest buffer overflows. Whenever the heap is large relative to the maximum amount of live memory, the likelihood of masking an error increases. For example, when the heap is no more than $1/8$ full, DieHard in stand-alone mode provides an 87.5% chance of masking 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.

6.2 Masking Dangling Pointers

A dangling pointer error occurs when an object is freed prematurely and its contents are overwritten by another object. 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 2. *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:*

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

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. Using the default configuration, the stand-alone version of DieHard has greater than a 99.5% chance of masking an 8-byte object that was freed 10,000 allocations too soon. Figure 4(b) shows the probabilities of avoiding dangling pointer errors for different object sizes and numbers of intervening allocations.

6.3 Detecting uninitialized reads

We say that DieHard detects an uninitialized read when it 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:*

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

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 paper, 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 [37] running reference workloads, as well as a suite of allocation-intensive benchmarks. These benchmarks perform between 100,000 and 1,700,000 memory operations per second (see Berger, Zorn and McKinley [6] for a detailed description). We include these benchmarks both because they are widely used in memory management studies [5, 17, 22] and because their unusually high allocation-intensity stresses memory management performance.

In all of our experiments, we set the default heap size for DieHard to 384MB, where up to 1/2 is available for allocation. This is larger than necessary for nearly all of the applications we measure here, but ensures consistency in our results. We also disable the replacement of unsafe library functions (see Section 4.4) for these experiments to isolate the protection that randomization and replication provide.

7.2 Overhead

We run our benchmarks on three different platforms: Linux, Windows XP, 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 Windows XP platform is a Pentium 4 system running at 3.20GHz with a 512K L2 cache and 2 gigabytes of RAM. All code on Windows is compiled using Visual Studio 7. 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 five runs after one warm-up run; observed variances are below 1%.

7.2.1 Linux

On Linux, we compare both DieHard and the Boehm-Demers-Weiser collector to the default GNU libc allocator, a variant of the Lea allocator [25]. The Boehm-Demers-Weiser collector is used for comparison because it represents an alternative trade-off in the design space between space, execution time, and safety guarantees. Figure 5(a) shows that, for the allocation-intensive benchmarks, DieHard suffers a performance penalty ranging from 16.5% to 63% (geometric mean: 40%). Its overhead is thus somewhat higher than that suffered by the Boehm-Demers-Weiser collector (2% to 59.7%, geometric mean 25.8%).

However, DieHard's runtime overhead is substantially lower for most of the SPECint2000 benchmarks. The geometric mean of DieHard's overhead is 12%. DieHard degrades performance substantially for two applications: 253.perlbmk (48.8%) and 300.twolf (109%). The 253.perlbmk benchmark is allocation-intensive, spending around 12.5% of its execution doing memory operations, highlighting both DieHard's and Boehm-Demers-Weiser's runtime overhead (13.4%). However, the 300.twolf 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 many size class partitions.

7.2.2 Windows

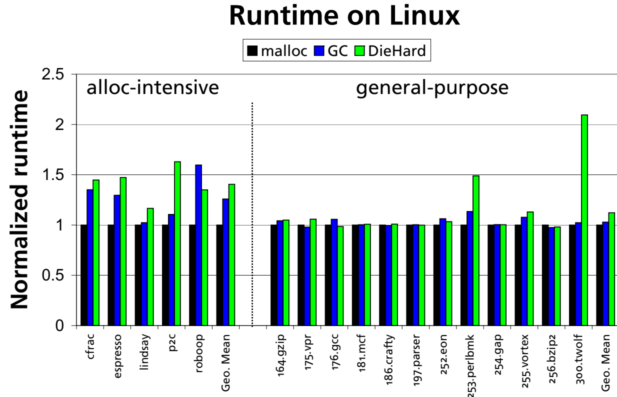
To evaluate the effect of different default allocators and compilers on DieHard's overhead, we ran the allocation-intensive benchmarks on Windows XP. Figure 5(b) presents execution time results for these benchmarks.

The results on Windows XP are far different than for Linux: the geometric mean of performance for these allocation-intensive benchmarks with DieHard is effectively the same as with the default allocator. DieHard *improves* runtime performance for roboop by 19%, espresso by 8.2%, and cfrac by 6.4%. Only lindsay and p2c perform slower, by 13.6% and 22.5% respectively.

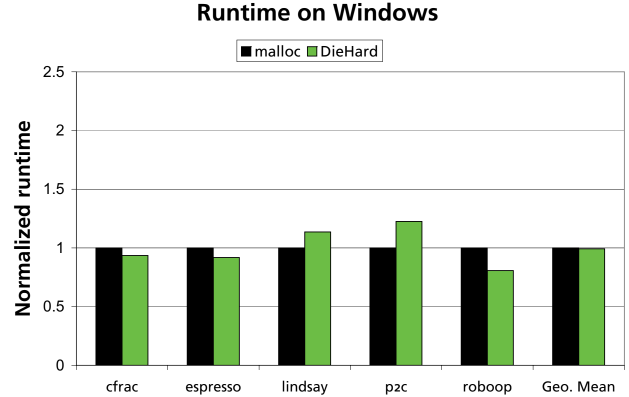
We attribute these results to two factors: first, the default Windows XP allocator is substantially slower than the Lea allocator. Second, Visual Studio produces much faster code for DieHard than g++ does. DieHard is written in modular C++ code with many small methods, and Visual Studio's better inlining heuristics and backend code generator combine to substantially improve DieHard's performance.

7.2.3 Solaris: Replicated Experiments

To quantify the overhead of the replicated framework and verify its scalability, we measure running time with sixteen replicas on a



(a) Linux: Performance of the default `malloc`, the Boehm-Demers-Weiser garbage collector, and DieHard (stand-alone version), across a range of allocation-intensive and general-purpose benchmark applications.



(b) Windows XP: Performance of the default `malloc` and DieHard (stand-alone version), across a range of allocation-intensive benchmark applications.

Figure 5. Runtime performance on Linux and Windows XP.

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 a real application, the Squid web caching server.

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 underflow, 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 the espresso benchmark, and running it ten times with the default allocator and with DieHard. We first introduced dangling pointers of frequency of 50% with distance 10: one out of every two objects is freed ten allocations too early. This high error rate prevents espresso from running to completion with the default allocator in all runs. However, with DieHard, espresso runs correctly in 9 out of 10 runs.

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 in 9 out of 10 runs and enters an infinite loop in the tenth. With DieHard, it runs successfully in all 10 of 10 runs.

Real Faults

We also tested DieHard on an actual buggy application. 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.

8. Related Work

This section describes related work in software engineering, fault tolerance, memory management, approaches to address security vulnerabilities, failure masking, fail-stop, debugging and testing. Table 1 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 [2]. 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 [10]. 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.

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 [25, 33], while others (Rockall [4], dnmalloc [43]) 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 achieve reasonable performance

Error	GNU libc [25]	BDW GC [9]	CCured [27]	Rx [30]	Failure-oblivious [32]	DieHard
heap metadata overwrites	undefined	undefined	abort	✓	undefined	✓
invalid frees	undefined	✓	✓	undefined	undefined	✓
double frees	undefined	✓	✓	✓	undefined	✓
dangling pointers	undefined	✓	✓	undefined	undefined	✓*
buffer overflows	undefined	undefined	abort	undefined	undefined	✓*
uninitialized reads	undefined	undefined	abort	undefined	undefined	abort*

Table 1. This table compares how various systems handle memory safety errors: ✓denotes correct execution, `undefined` 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.

(3X-5X more than `malloc/free`) [20, 38, 44]. 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 [7, 29] or randomly pad allocation requests [8]. Base address randomization provides little protection from heap-based attacks on 32-bit platforms [36]. 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.

Failure masking: Several researchers have proposed unsound techniques that can prevent programs from crashing [13, 30, 32]. *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 [13]. 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 [32]. 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 [30]. 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.

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 [1, 3, 27, 41, 42]. 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 [21]. 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 [27]. While Cyclone uses region-based memory management and safe explicit deallocation [16, 38], 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 [18] and Valgrind [28] 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 [19] uses sampling to detect memory leaks at runtime with little overhead (around 5%), and could be employed in conjunction with DieHard.

9. Conclusion

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 on Linux DieHard, adds little CPU overhead to many of the SPECint2000 benchmark programs, while the CPU overhead in allocation-intensive programs is larger. On Windows, the overhead of DieHard is reduced, and programs with DieHard occasionally run faster than when running with the default allocator.

We show analytically that DieHard increases error tolerance, and reaffirm our analytic results by demonstrating that DieHard significantly increases the error tolerance of an application in which faults are artifically injected. We also describe an experiment in which DieHard successfully tolerates a known buffer-overflow error in the Squid web cache server.

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.

The current implementation of DieHard has limitations that we believe can be overcome. The DieHard algorithm as implemented initializes the heap based on the maximum size the heap will eventually grow to. We plan to investigate an adaptive version of DieHard that grows memory regions dynamically as objects are allocated. Other ways of reducing the memory requirements of DieHard include selectively applying the technique to particular size classes, allocation pools, object types, and/or object instances.

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.) In the future, we hope to better characterize program output so that these kinds of irreproducible results can be recognized and factored.

Beyond error tolerance, DieHard also can be used to debug memory corruption. By differencing the heaps of correct and incorrect executions of applications, it may be possible to pinpoint the exact locations of memory errors and report these as part of a crash dump without the crash.

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 paper, 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, 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.

This material is based upon work supported by the National Science Foundation under CAREER Award CNS-0347339. 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.

DieHard is publicly available at http://www.cs.umass.edu/~emery/diehard/.

References

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

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

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

[4] T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *7th International Conference on Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 158–173, 2001.

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

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

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

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

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

[10] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 1–11, New York, NY, USA, 1995. ACM Press.

[11] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 1–12, Atlanta, May 1999. ACM Press.

[12] D. L. Detlefs. Empirical evidence for using garbage collection in C and C++ programs. In E. Moss, P. R. Wilson, and B. Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, Oct. 1993.

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

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

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[16] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293, New York, NY, USA, 2002. ACM Press.

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

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

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

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

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

[22] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In P. Dickman and P. R. Wilson, editors, *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, Oct. 1997.

[23] M. Kaempf. Vudo malloc tricks. *Phrack Magazine*, 57(8), Aug. 2001.

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

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

[26] G. Marsaglia. yet another RNG. posted to the electronic bulletin board sci.stat.math, Aug. 1994.

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

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

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

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

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

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

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

[34] J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM*, 21(3):419–499, July 1974.

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

[36] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and Communications Security*, pages 298–307, New York, NY, USA, 2004. ACM Press.

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

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

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

[40] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116, Kinross, Scotland, Sept. 1995. Springer-Verlag.

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

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

[43] Y. Younan, W. Joosen, F. Piessens, and H. V. den Eynden. Security of memory allocators for C and C++. Technical Report 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.

[44] B. Zorn. The measured cost of conservative garbage collection. *Software Practice and Experience*, 23:733–756, 1993.