Exterminator: Automatically Correcting Memory Errors with High Probability

Gene Novark
Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
gnovark@cs.umass.edu

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

Programs written in C and C++ are susceptible to memory errors, including buffer overflows and dangling pointers. These errors, which can lead to crashes, erroneous execution, and security vulnerabilities, are notoriously costly to repair. Tracking down their location in the source code is difficult, even when the full memory state of the program is available. Once the errors are finally found, fixing them remains challenging: even for critical security-sensitive bugs, the average time between initial reports and the issuance of a patch is nearly one month.

We present Exterminator, a system that automatically corrects heap-based memory errors without programmer intervention. Exterminator exploits randomization to pinpoint errors with high precision. From this information, Exterminator derives **runtime patches** that fix these errors both in current and subsequent executions. In addition, Exterminator enables collaborative bug correction by merging patches generated by multiple users. We present analytical and empirical results that demonstrate Exterminator's effectiveness at detecting and correcting both injected and real faults.

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

General Terms Algorithms, Languages, Reliability, Security

Keywords DieFast, Exterminator, dynamic memory allocation, error correction, memory errors, probabilistic memory safety, randomized algorithms

1. Introduction

The use of manual memory management and unchecked memory accesses in C and C++ leaves applications written in these languages susceptible to a range of memory errors. These include buffer overruns, where reads or writes go beyond allocated regions, and dangling pointers, when a program deallocates memory while it is still live. Memory errors can cause programs to crash or produce incorrect results. Worse, attackers are frequently able to exploit these memory errors to gain unauthorized access to systems.

Debugging memory errors is notoriously difficult and time-consuming. Reproducing the error requires an input that exposes it. Since inputs are often unavailable from deployed programs, developers must either concoct such an input or find the problem via code inspection. Once a test input is available, software developers typically execute the application with heap debugging tools like Purify [21] and Valgrind [30, 40], which slow execution by an order of magnitude. When the bug is ultimately discovered, developers must construct and carefully test a patch to ensure that it fixes the bug without introducing any new ones. According to Symantec, the average time between the discovery of a critical, *remotely exploitable* memory error and the release of a patch for enterprise applications is 28 days [44].

As an alternative to debugging memory errors, researchers have proposed a number of systems that either detect or tolerate them. *Fail-stop* systems are compiler-based approaches that require access to source code, and abort programs when they performs illegal operations like buffer overflows [1, 2, 14, 16, 29, 45, 46]. They rely either on conservative garbage collection [9] or pool allocation [15, 17] to prevent or detect dangling pointer errors. *Failure-oblivious* systems are also compiler-based, but manufacture read values and drop or cache illegal writes for later reuse [35, 36]. Finally, *fault-tolerant* systems mask the effect of errors, either by logging and replaying inputs in an environment that pads allocation requests and defers deallocations (e.g., Rx [32]), or through randomization and optional voting-based replication that reduces the odds that an error will have any effect (e.g., DieHard [3]).

Contributions: This paper presents **Exterminator**, a runtime system that not only tolerates but also detects and corrects heap-based memory errors. Exterminator requires neither source code nor programmer intervention, and fixes existing errors without introducing new ones. To our knowledge, this system is the first of its kind.

Exterminator relies on an efficient probabilistic debugging allocator that we call **DieFast**. DieFast is based on DieHard's allocator [3, 4], which ensures that heaps are independently randomized. However, while DieHard can only probabilistically tolerate errors, DieFast probabilistically detects them.

When Exterminator discovers an error, it dumps a **heap image** that contains the complete state of the heap. Exterminator's **probabilistic error isolation** algorithm then processes one or more heap images to locate the source and size of buffer overflows and dangling pointer errors. This error isolation algorithm has provably low false positive and false negative rates.

Once Exterminator locates a buffer overflow, it determines the allocation site of the overflowed object, and the size of the over-

PLDI'07 June 11–13, 2007, San Diego, California, USA.

Error	DieHard [3]	Exterminator
invalid frees	tolerate	tolerate
double frees	tolerate	tolerate
uninitialized reads	detect*	N/A
dangling pointers	tolerate*	tolerate* & **correct***
buffer overflows	tolerate*	tolerate* & **correct***

Table 1. A summary of how Exterminator handles particular memory errors (Section 2): invalid and double frees have no effect, and Exterminator probabilistically corrects dangling pointers and buffer overflows. The asterisk superscript means "probabilistically."

flow. For dangling pointer errors, Exterminator determines both the allocation and deletion sites of the dangled object, and computes how prematurely the object was freed.

With this information in hand, Exterminator corrects the errors by generating **runtime patches**. These patches operate in the context of a **correcting allocator**. The correcting allocator prevents overflows by padding objects, and prevents dangling pointer errors by deferring object deallocations. These actions impose little space overhead because Exterminator's runtime patches are tailored to the specific allocation and deallocation sites of each error.

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

Exterminator can operate in three distinct modes: an **iterative mode** for runs over the same input, a **replicated mode** that can correct errors on-the-fly, and a **cumulative mode** that corrects errors across multiple runs of the same application.

We experimentally demonstrate that, in exchange for modest runtime overhead (geometric mean of 25%), Exterminator effectively isolates and corrects both injected and real memory errors, including buffer overflows in the Squid web caching server and the Mozilla web browser.

Outline: The remainder of this paper is organized as follows. First, Section 2 describes the errors that Exterminator detects and corrects. Next, Section 3 introduces Exterminator's software architecture. Section 4 presents Exterminator's error isolation algorithms for its iterative and replicated modes, and Section 5 describes the isolation algorithms for its cumulative mode. Section 6 describes the correction algorithm that applies the patches that the error isolator generates. Section 7 empirically evaluates their cost and effectiveness on real applications, both with injected and actual memory errors. Finally, Section 8 discusses key related work, Section 9 presents directions for future work, and Section 10 concludes.

2. Memory Errors

Table 1 summarizes the memory errors that Exterminator addresses, and its response to each. Exterminator identifies and corrects *dangling pointers*, where a heap object is freed while it is still live, and *buffer overflows* (a.k.a. buffer overruns) of heap objects. Notice that this differs substantially from DieHard, which tolerates these errors probabilistically but cannot detect or correct them.

Exterminator's allocator (DieFast) inherits from DieHard its immunity from two other common memory errors: *double frees*, when a heap object is deallocated multiple times without an intervening allocation, and *invalid frees*, when a program deallocates an object that was never returned by the allocator. These errors have serious consequences in other systems, where they can lead to heap corruption or abrupt program termination.

Exterminator prevents these invalid deallocation requests from having any impact. DieFast's bitmap-based allocator (Section 3.2) makes multiple frees benign since a bit can only be reset once. By checking ranges, DieFast detects and ignores invalid frees.

Figure 1. An abstract view of Exterminator's heap layout. Metadata below the horizontal line contains information used for error isolation and correction (see Section 3.2).

2.1 Limitations

Exterminator's ability to correct both dangling pointer errors and buffer overflows has several limitations. First, Exterminator assumes that buffer overflows always corrupt memory at higher addresses—that is, they are *forward* overflows. While it is possible to extend Exterminator to handle backwards overflows, we have not implemented this functionality. Exterminator can only correct finite overflows, so that it can contain any given overflow by overallocation. Similarly, Exterminator corrects dangling pointer errors by inserting finite delays before freeing particular objects. Finally, in iterated and replicated modes, Exterminator assumes that overflows and dangling pointer errors are deterministic. However, the cumulative mode does not require deterministic errors.

Unlike DieHard, Exterminator does not detect *uninitialized reads*, where a program makes use of a value left over in a previously-allocated object. Because the intended value is unknown, it is not generally possible to repair such errors without additional information, e.g. data structure invariants [12]. Instead, Exterminator fills all allocated objects with zeroes.

3. Software Architecture

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

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

3.1 DieHard Overview

The DieHard system includes a bitmap-based, fully-randomized memory allocator that provides *probabilistic memory safety* [3]. The latest version of DieHard, upon which Exterminator is based, adaptively sizes its heap be M times larger than the maximum needed by the application [4] (see Figure 2). This version of DieHard allocates memory from increasingly large chunks that we call *miniheaps*. Each miniheap contains objects of exactly one size. If an allocation would cause the total number of objects to exceed $1/M$, DieHard allocates a new miniheap that is twice as large as the previous largest miniheap.

Allocation randomly probes a miniheap's bitmap for the given size class for a free bit: this operation takes $O(1)$ expected time. Freeing a valid object resets the appropriate bit. DieHard's use of randomization across an over-provisioned heap makes it probabilistically likely that buffer overflows will land on free space, and unlikely that a recently-freed object will be reused soon, making dangling pointer errors rare.

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

3.2 Exterminator's Heap Layout

Figure 1 presents Exterminator's heap layout, which includes five fields per object for error isolation and correction: an **object id**, **allocation** and **deallocation sites**, **deallocation time**, which records when the object was freed, and a **canary bitset** that indicates if the object was filled with canaries (Section 3.3).

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

Figure 3. Site information hash function, used to store allocation and deallocation call sites (see Section 3.2).

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

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

Figure 4. Pseudo-code for DieFast, a probabilistic debugging allocator (Section 3.3).

An object id of n means that the object is the nth object allocated. Exterminator uses object ids to identify objects across multiple heaps. These ids are needed because the object's address cannot be used to identify it across differently-randomized heaps.

The site information fields capture the calling context for allocations and deallocations. For each, Exterminator hashes the least significant bytes of the five most-recent return addresses into 32 bits using the DJB2 hash function [6] (see Figure 3).

This out-of-band metadata accounts for approximately 16 bytes plus two bits of space overhead for every object. This overhead is comparable to that of typical freelist-based memory managers like the Lea allocator, which prepend 8-byte (on 32-bit systems) or 16-byte headers (on 64-bit systems) to allocated objects [24].

3.3 DieFast: A Probabilistic Debugging Allocator

Exterminator uses a new, probabilistic debugging allocator that we call DieFast. DieFast uses the same randomized heap layout as DieHard, but extends its allocation and deallocation algorithms to detect and expose errors. Figure 4 presents pseudo-code for the DieFast allocator. Unlike previous debugging allocators, DieFast has a number of unusual characteristics tailored for its use in the context of Exterminator.

Implicit Fence-posts

Many existing debugging allocators pad allocated objects with fence-posts (filled with **canary** values) on both sides. They can thus detect buffer overflows by checking the integrity of these fence-

posts. This approach has the disadvantage of increasing space requirements. Combined with the already-increased space requirements of a DieHard-based heap, the additional space overhead of padding may be unacceptably large.

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

Random Canaries

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

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

Probabilistic Fence-posts

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

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

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

In cumulative mode, Exterminator prevents this scenario by non-deterministically writing canaries into freed memory randomly with probability p, and setting the appropriate bit in the canary bitmap. This probabilistic approach may seem to degrade Exterminator's ability to find errors. However, it is required to isolate read-only dangling pointer errors, where the canary itself remains intact. Because it would take an impractically large number of iterations or replicas to isolate these errors, Exterminator always fills freed objects with canaries when not running in cumulative mode (see Sections 5.2 and 7.2 for discussion).

Probabilistic Error Detection

Whenever DieFast allocates memory, it examines the memory to be returned to verify that any canaries are intact. If not, in addition to signalling an error (see Section 3.4), DieFast sets the allocated bit for this chunk of memory. This "bad object isolation" ensures that the object will not be reused for future allocations, preserving its contents for Exterminator's subsequent use. Checking canary integrity on each allocation ensures that DieFast will detect heap corruption within $E(H)$ allocations, where H is the number of objects on the heap.

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

3.4 Modes of Operation

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

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

Iterative Mode

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

Once it reaches the malloc breakpoint, Exterminator triggers another heap image dump. This process can be repeated multiple times to generate independent heap images. Exterminator then performs post-mortem error isolation and runtime patch generation. A small number of iterations usually suffices for Exterminator to generate runtime patches for an individual error, as we show in Section 7.2. When run with a correcting memory allocator that incorporates these changes (described in detail in Section 6.3), these patches automatically fix the isolated errors.

Replicated Mode

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

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

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

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

Figure 5. Exterminator's replicated architecture (Section 3.4). Replicas are equipped with different seeds that fully randomize their DieFast-based heaps (Section 3.3), input is broadcast to all replicas, and output goes to a voter. A crash, output divergence, or signal from DieFast triggers the error isolator (Section 4), which generates *runtime patches*. These patches are fed to correcting allocators (Section 6), which fix the bug for current and subsequent executions.

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

Cumulative Mode

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

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

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

4. Iterative and Replicated Error Isolation

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

When operating in its replicated or iterative modes, Exterminator's probabilistic error isolation algorithm operates by searching for discrepancies across multiple heap images. Exterminator relies on corrupted canaries to indicate the presence of an error. A corrupted canary (one that has been overwritten) can mean two things: if every object has the same corruption, then it is likely a dangling pointer error, as Theorem 1 shows. If canaries are corrupted in multiple objects, then it is likely to be a buffer overflow. Exterminator limits the number of false positives for both overflows and dangling pointer errors.

4.1 Buffer Overflow Detection

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

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

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

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

Finally, an object can contain values that legitimately differ from process to process. Examples of these values include process ids, file handles, pseudorandom numbers, and pointers in data structures that depend on addresses (e.g., some red-black tree implementations). When Exterminator examines an object and encounters any word that differs at the same position across the heaps, it considers it to be legitimately different, and not an overflow.

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

Theorem 1. *Let k be the number of heap images, S the length (*in number of objects*) of the* **overflow string**, *and H the number of objects on the heap. Then the probability of an overflow overwriting k objects identically is at most:*

$$P(\text{identical overflow}) \quad \leq \quad \frac{1}{2^k} \times \frac{1}{(H-S)^k}.$$

Proof. Assume that buffer overflows overwrite past the end of an object. Thus, for an overflow from object i to land on a given object j, it must both precede it and be large enough to span the distance from i to j. An object i precedes j in k heaps with probability $(1/2)^k$. Objects i and j are separated by S or fewer objects with probability at most $(1/(H-S))^k$. Combining these terms yields the above formula. □

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

Theorem 2. *Let M be the heap multiplier, so a heap is never more than $1/M$ full. The likelihood that an overflow of length b bytes fails to be detected by comparison against a canary is at most:*

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

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

Culprit Identification

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

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

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

Theorem 3. *The expected number of objects (possible culprits) the same distance δ from any given victim object across k heaps is:*

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

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

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

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

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

4.2 Dangling Pointer Isolation

Isolating dangling pointer errors falls into two cases: a program may *read and write* to the dangled object, leaving it partially or completely overwritten, or it may only *read* through the dangling pointer. Exterminator does not handle read-only dangling pointer errors in iterative or replicated mode because it would require too many replicas (e.g., around 20; see Section 7.2). However, it handles overwritten dangling objects straightforwardly.

When a freed object is overwritten with identical values across multiple heap images, Exterminator classifies the error as a dangling pointer overwrite. As Theorem 1 shows, this situation is highly unlikely to occur for a buffer overflow. Exterminator then generates an appropriate runtime patch, as Section 6.2 describes.

5. Cumulative Error Isolation

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

5.1 Buffer Overflow Detection

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

After computing the set of corrupt object slots, Exterminator examines allocation sites and finds possible culprits. To reason about an individual allocation site, Exterminator must consider all observed objects from that site.

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

For each object, the error isolation algorithm computes the probability that the object satisfies these criteria. The total probability is the product of the probabilities of being allocated in the same miniheap (the left-hand term below), times the probability of it falling on the left side of the corruption (the right-hand term). The first term is the size of the corrupt miniheap, divided by the sum of the sizes of all miniheaps available in the size class at the time the object was allocated. Let M_c be the corrupted miniheap, k the index of the corrupted slot in M_c, $\tau(i)$ and $\tau(M_j)$ the allocation time of object i or miniheap M_j, respectively, and $size(M_i)$ the number of object slots in miniheap M_i. The probability $P(C_i)$ that object i satisfies the criteria is then:

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

where

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

For each allocation site A, Exterminator then computes the probability $P(C_A)$ that at least one object from the site satisfied the

criteria (1 minus the probability of all objects *not* satisfying) as

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

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

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

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

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

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

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

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

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

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

The likelihood is then:

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

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

5.2 Dangling Pointer Isolation

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

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

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

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

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

6. Error Correction

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

6.1 Buffer overflow correction

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

6.2 Dangling pointer correction

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

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

This choice ensures that Exterminator will compute a correct patch in a logarithmic number of executions. As we show in Section 7.2, multiple iterations to correct pointer errors are rare in practice, because the last allocation time can be well past the time that the object should have been freed.

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

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

Figure 6. Pseudo-code for the correcting memory allocator, which incorporates the runtime patches generated by the error isolator.

It is important to note that this deallocation deferral does not multiply its lifetime but rather its *drag* [39]. To illustrate, an object might live for 1000 allocations and then be freed just 10 allocations too soon. If the program immediately crashes, Exterminator will extend its lifetime by 21 allocations, increasing its lifetime by less than 1% (1021/1010). Section 7.3 evaluates the impact of both overflow and dangling pointer correction on space consumption.

6.3 The Correcting Memory Allocator

The correcting memory allocator incorporates the runtime patches described above and applies them when appropriate. Figure 6 presents pseudo-code for the allocation and deallocation functions.

At start-up, or upon receiving a reload signal (Section 3.4), the correcting allocator loads the runtime patches from a specified file. It builds two hash tables: a **pad table** mapping allocation sites to pad sizes, and a **deferral table**, mapping pairs of allocation and deallocation sites to a deferral value. Because it can reload the runtime patch file and rebuild these tables on-the-fly, Exterminator can apply patches to running programs without interrupting their execution. This aspect of Exterminator's operation may be especially useful for systems that must be kept running continuously.

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

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

Figure 7. Runtime overhead for Exterminator across a suite of benchmarks, normalized to the performance of GNU libc (Linux) allocator.

6.4 Collaborative Correction

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

7. Results

Our evaluation answers the following questions: (1) What is the runtime overhead of using Exterminator? (2) How effective is Exterminator at finding and correcting memory errors, both for injected and real faults? (3) What is the overhead of Exterminator's runtime patches?

7.1 Exterminator Runtime Overhead

We evaluate Exterminator's performance with the SPECint2000 suite [43] running reference workloads, as well as a suite of allocation-intensive benchmarks. We use the latter suite of benchmarks both because they are widely used in memory management studies [3, 19, 22], and because their high allocation-intensity stresses memory management performance. For all experiments, we fix Exterminator's heap multiplier (value of M) at 2.

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

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

We compare the runtime of Exterminator (DieFast plus the correcting allocator) to the GNU libc allocator. This allocator is based on the Lea allocator [24], which is among the fastest available [5]. Figure 7 shows that, versus this allocator, Exterminator degrades performance by from 0% (`186.crafty`) to 132% (`cfrac`), with a geometric mean of 25.1%. While Exterminator's overhead is substantial for the allocation-intensive suite (geometric mean: 81.2%), where the cost of computing allocation and deallocation contexts dominates, its overhead is significantly less pronounced across the SPEC benchmarks (geometric mean: 7.2%).

7.2 Memory Error Correction

Injected Faults

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

Buffer overflows: We triggered 10 different buffer overflows each of three different sizes (4, 20, and 36 bytes) by underflowing objects in the `espresso` benchmark. The number of images required to isolate and correct these errors was 3 in every case. Notice that this result is substantially better than the analytical worst-case. For three images, Theorem 2 bounds the worst-case likelihood of missing an overflow to 42% (Section 4.1), rather than the 0% false negative rate we observe here.

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

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

Real Faults

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

Squid web cache: Version 2.3s5 of Squid has a buffer overflow; certain inputs cause Squid to crash with either the GNU libc allocator or the Boehm-Demers-Weiser collector [3, 32].

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

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

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

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

7.3 Patch Overhead

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

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

8. Related Work

8.1 Randomized Memory Managers

Several memory management systems employ some degree of randomization, including locating the heap at a random base address [7, 31], adding random padding to allocated objects [8], shuffling recently-freed objects [23], or a mix of padding and object deferral [32]. This level of randomization is insufficient for Exterminator, which requires full heap randomization.

Exterminator builds on DieHard, which tolerates errors probabilistically; Section 3.1 provides an overview. Exterminator substantially modifies and extends DieHard's heap layout and allocation algorithms. It also uses probabilistic algorithms that identify and correct errors.

8.2 Automatic Repair

Demsky et al.'s *automatic data structure repair* [11, 12, 13] enforces data structure consistency specifications, guided by a formal

description of the program's data structures (specified manually or derived automatically by Daikon [18]). Exterminator attacks a different problem, namely that of isolating and correcting memory errors, and is orthogonal and complementary to data structure repair.

Sidiroglou et al. propose STEM, a self-healing runtime that executes functions in a transactional environment so that if they detect the function misbehaving, they can prevent it from doing damage [41]. Using STEM, they implement error virtualization, which maps the set of possible errors in a function onto those that have an explicit error handler. The more recent SEAD system goes beyond STEM requiring no source code changes, handling I/O with virtual proxies, and by specifying the repair policy explicitly through an external description [42]. While STEM and SEAD are promising approaches to automatically recovering from errors, neither provides solutions for as broad a class of errors as Exterminator, nor do they provide mechanisms to semantically eliminate the source of the error automatically, as Exterminator does.

8.3 Automatic Debugging

Two previous systems apply techniques designed to help isolate bugs. *Statistical bug isolation* is a distributed assertion sampling technique that helps pinpoint the location of errors, including but not limited to memory errors [25, 26, 27]. It works by injecting lightweight tests into the source code; the result of these tests, in bit vector form, can be processed to generate likely sources of the errors. This statistical processing differs from Exterminator's probabilistic error isolation algorithms, although Liu et al. also use hypothesis testing [27]. Like statistical bug isolation, Exterminator can leverage the runs of deployed programs. However, unlike statistical bug isolation, Exterminator requires neither source code nor a large deployed user base in order to find errors, and automatically generates runtime patches that correct them.

Delta debugging automates the process of identifying the smallest possible inputs that do and do not exhibit a given error [10, 28, 48]. Given these inputs, it is up to the software developer to actually locate the bugs themselves. Exterminator focuses on a narrower class of errors, but is able to isolate and correct an error given just one erroneous input, regardless of its size.

8.4 Fault Tolerance

Recently, there has been an increasing focus on approaches for tolerating hardware transient errors that are becoming more common due to fabrication process limitations. Work in this area ranges from proposed hardware support [33] to software fault tolerance [34]. While Exterminator also uses redundancy as a method for detecting and correcting errors, Exterminator goes beyond tolerating software errors, which are not transient, to correcting them permanently. Like Exterminator, other efforts in the fault tolerance community seek to gather data from multiple program executions to identify potential errors. For example, Guo et al. use statistical techniques on internal monitoring data to probabilistically detect faults, including memory leaks and deadlocks [20]. Exterminator goes beyond this previous work by characterizing each memory error so specifically that a correction can be automatically generated for it.

Rinard et al. present a compiler-based approach called *boundless buffers* that caches out-of-bound writes in a hash table for later reuse [35]. 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).

Rx operates by checkpointing program execution and logging inputs [32]. Rx rolls back crashed applications and replays inputs to it in a new environment that pads all allocations or defers all deallocations by some amount. If this new environment does not yield success, Rx rolls back the application again and increases the pad values, up to some threshold. Unlike Rx, Exterminator does not require checkpointing or rollback, and precisely isolates and corrects memory errors.

8.5 Memory Managers

Conservative garbage collection prevents dangling pointer errors [9], but does not prevent buffer overflows. Exterminator's error isolation and correction is orthogonal to garbage collection.

Finally, there have been numerous debugging memory allocators; the documentation for one of them, *mpatrol*, includes a list of over ninety such systems [38]. Notable recent allocators with debugging features include dnmalloc [47], Heap Server [23], and version 2.8 of the Lea allocator [24, 37]. Exterminator either prevents or corrects errors that these allocators can only detect.

9. Future Work

While Exterminator can effectively locate and correct memory errors on the heap, it does not yet address stack errors. We are investigating approaches to apply Exterminator to the stack.

In addition, while Exterminator's runtime patches contain information that describe the error location and its extent, it is not in a human-readable form. We plan to develop a tool to process runtime patches into bug reports with suggested fixes.

10. Conclusion

This paper presents Exterminator, a system that automatically corrects heap-based memory errors in C and C++ programs with high probability. Exterminator operates entirely at the runtime level on unaltered binaries, and consists of three key components: (1) DieFast, a probabilistic debugging allocator, (2) a probabilistic error isolation algorithm, and (3) a correcting memory allocator. Exterminator's probabilistic error isolation isolates the source and extent of memory errors with provably low false positive and false negative rates. Its correcting memory allocator incorporates runtime patches that the error isolation algorithm generates to correct memory errors. Exterminator is not only suitable for use during testing, but also can automatically correct deployed programs.

Acknowledgments

The authors would like to thank Sam Guyer, Mike Hicks, Erik Learned-Miller, Sarah Osentoski, Martin Rinard, and the anonymous reviewers for their valuable feedback. This material is based upon work supported by Intel, Microsoft Research, and the National Science Foundation under CAREER Award CNS-0347339 and CNS-0615211. 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.

References

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

[2] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering*, pages 332–341, New York, NY, USA, 2005. ACM Press.

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

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

[5] 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*, Snowbird, Utah, June 2001.

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

[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] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, pages 342–351, 2005.

[11] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 233–244, New York, NY, USA, 2006. ACM Press.

[12] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, pages 78–95, New York, NY, USA, 2003. ACM Press.

[13] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 27th International Conference on Software Engineering*, pages 176–185, 2005.

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

[15] D. Dhurjati and V. Adve. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 269–280, 2006.

[16] D. Dhurjati, S. Kowshik, and V. Adve. SAFEcode: enforcing alias analysis for weakly typed languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 144–157, New York, NY, USA, 2006. ACM Press.

[17] 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.

[18] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, New York, NY, USA, 2000. ACM Press.

[19] 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.

[20] Z. Guo, G. Jiang, H. Chen, and K. Yoshihira. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks*, pages 259–268, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[21] 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.

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

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

[25] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

[26] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 15–26, New York, NY, USA, 2005. ACM Press.

[27] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 286–295, New York, NY, USA, 2005. ACM Press.

[28] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, pages 142–151, New York, NY, USA, 2006. ACM Press.

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

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

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

[32] 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.

[33] M. K. Qureshi, O. Mutlu, and Y. N. Patt. Microarchitecture-based introspection: a technique for transient-fault tolerance in microprocessors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN 2005)*, pages 434–443, 2005.

[34] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[35] 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.

[36] 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.

[37] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time

detection of heap-based overflows. In *Proceedings of the 17th Large Installation Systems Administration Conference*, pages 51–60. USENIX, 2003.

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

[39] C. Runciman and N. Rojemo. Lag, drag and postmortem heap profiling. In *Implementation of Functional Languages Workshop*, Bastad, Sweden, Sept. 1995.

[40] 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.

[41] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *USENIX Annual Technical Conference*, pages 149–161. USENIX, 2005.

[42] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. From STEM to SEAD: Speculative execution for automated defense. In *USENIX Annual Technical Conference*. USENIX, 2007.

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

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

[45] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, pages 117–126, New York, NY, USA, 2004. ACM Press.

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

[47] 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.

[48] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.