Exterminator: Automatically Correcting Memory Errors

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 and replication to pinpoint errors with high precision. From this information, Exterminator derives *runtime patches* that fix these errors in current and subsequent executions. In addition, Exterminator enables collaborative bug repair by merging patches generated by multiple users. We present analytical and empirical results that demonstrate Exterminator's effectiveness at detecting heap errors and correcting them, for both injected and real faults.

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 [20] and Valgrind [29, 38], 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 [40].

As an alternative to debugging memory errors, researchers have proposed a number of systems that either detect or tolerate them. *Fail-stop* systems are typically compiler-based approaches that require access to source code, and abort programs when they performs illegal operations like buffer overflows [1, 2, 13, 15, 28, 41, 42]. They rely either on conservative garbage collection [7] or pool allocation [14, 16] to prevent or detect dangling pointer errors. *Failure-oblivious* systems employ similar compiler technology to detect overruns, but instead of aborting, they manufacture read values and drop or cache illegal writes for later reuse [34]. Finally, *fault-tolerant* systems mask the effect of errors, either by logging and replaying inputs in a new execution environment that pads allocation requests and defers deallocations (e.g., Rx [31]), or through randomization and voting-based replication that probabilistically reduces the odds that an error will have any effect (e.g., DieHard [3]).

These approaches mitigate but can not eliminate the effect of memory errors. In fail-stop systems, repeated memory errors become denial-of-service attacks. In failure-oblivious systems, they lead to increasingly unpredictable behavior. And although fault-tolerant systems may reduce the impact of errors, repeated errors eventually will cause these systems to fail.

Contributions: This paper presents **Exterminator**, a runtime system that both 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 operates either with replication or iteration. It relies on an efficient probabilistic debugging allocator that we call **DieFast**. DieFast is based on DieHard's allocator [3], which ensures that heaps are independently randomized. However, while DieHard can only probabilistically tolerate errors, DieFast probabilistically both detects and exposes errors when they occur.

When Exterminator discovers an error, it asynchronously dumps a **heap image** that contains the complete state of the heap. Exterminator's **probabilistic error isolation** algorithm then processes multiple heap images (from replicas or iterated runs) 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 overflow. For dangling pointer errors, Exterminator determines both the allocation and deletion sites of the dangled object, and computes how prematurely the object was freed.

Submitted for publication.

Error	DieHard [3]	Exterminator
invalid frees	tolerate	tolerate
double frees	tolerate	tolerate
uninitialized reads	detect*	detect*
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."

With this information in hand, Exterminator repairs 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 repair**: users running Exterminator can automatically merge their patches, thus systematically and continuously improving application reliability.

We experimentally demonstrate that, in exchange for modest runtime overhead (geometric mean of 24%), Exterminator effectively isolates and repairs both injected and real memory errors, including a buffer overflow in the Squid web caching server. Our prototype implementation of Exterminator currently targets single-threaded programs; Section 8.1 discusses our plans to extend Exterminator to work with multi-threaded applications.

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 then presents Exterminator's error isolation algorithm, and analytically quantifies its precision. Section 5 describes the repair algorithm that applies patches that the error isolator generates. Section 6 empirically evaluates their cost and effectiveness on real applications, both with injected and actual memory errors. Finally, Section 7 discusses key related work, and Section 8 concludes with directions for future work.

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 instead of detecting or correcting 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.[1] or abrupt program termination.[2]

[1] Windows, Solaris, and GNU libc (Linux) $\leq$ 2.7.X.

[2] GNU libc $\geq$ 2.8.

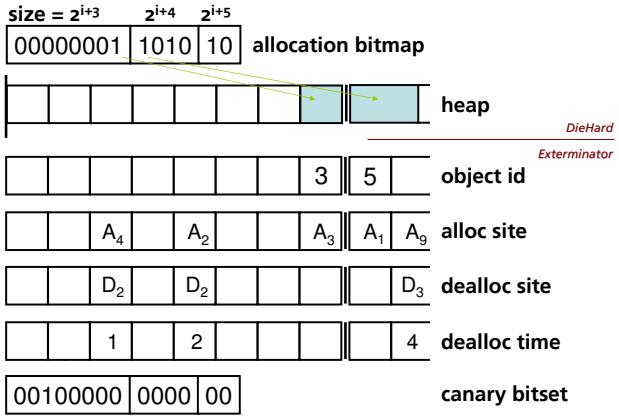

Figure 1. Exterminator's heap layout. Metadata below the horizontal line contains information used for error isolation and repair (see Section 3.2).

Exterminator detects and prevents these invalid deallocation requests from having any impact. DieFast's bitmap-based allocator (Section 3.2) makes multiple frees impossible since a bit can only be reset once. By checking alignment, DieFast detects and ignores invalid frees. Exterminator also probabilistically detects *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 [11].

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 both of Exterminator's modes of operation: an *iterative* mode intended for use during testing, and a *replicated* mode suitable for use by deployed applications.

3.1 DieHard Overview

Berger and Zorn's DieHard system includes a bitmap-based, fully-randomized memory allocator that provides *probabilistic memory safety* [3]. It uses a heap sized M times larger than the maximum needed for the application. Allocation randomly probes the bitmap associated with the given size class for a free bit (0): 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 can optionally use replication to further increase the probability of successful execution in the face of errors. In this case, it broadcasts inputs to a number of replicas, each of which is a copy of the application process equipped with a different random seed. A voter intercepts and compares outputs across the replicas, and only actually generates output agreed on by a plurality of the replicas. The independent randomization of each replica's heap makes the probabilities of memory errors independent. 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.

```
int computeHash (int * pc) {
  int hash = 0;
  for (int i = 0; i < 4; i++) {
    hash << 8;
    hash ^= pc[i];
  }
  return hash;
}
```

Figure 2. Site information hash function, used to store allocation and deallocation call sites (see Section 3.2).

3.2 Exterminator's Heap Layout

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

The object id is an integer specific to each size class that indicates which allocation this object corresponds to. An object id of 12 means that the object is the 12th object allocated from that size class. Exterminator uses object ids to identify objects across replicated heaps. These ids are needed because an object's address cannot be used to identify an object 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 four most-recent return addresses into a single 32-bit value (see Figure 2).

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 [23]. Exterminator's space overhead is fixed regardless of the underlying architecture.

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 3 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.

```
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) { 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; }
  // Fill with canary with P=1/2.
  if (random() < 0.5)
    fillWithCanary (ptr);
}
```

Figure 3. Pseudo-code for DieFast, a probabilistic debugging allocator (Section 3.3).

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 freed space.

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 only a $1/2^{32}$ possibility of a collision with the canary, thus ensuring a low false positive rate (see Theorem 2).

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 almost certainly trigger a segmentation violation. However, this approach is unsuitable for bug isolation.

The problem is that reading random values does not necessarily cause programs to fail immediately. 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.

To prevent this scenario, Exterminator non-deterministically writes canaries into freed memory randomly with probability $P = 1/2$, and sets the appropriate bit in the canary bitmap. While this probabilistic approach may seem to degrade Exterminator's

ability to find errors, it is in fact required to isolate read-only dangling pointer errors, as Section 4.2 describes.

Probabilistic Error Detection

Whenever DieFast allocates memory, it examines the value of the object to be returned to verify that its canaries are intact if it was filled with canaries. If so, DieFast returns the object. Otherwise, in addition to signalling an error (see Section 3.4), DieFast sets the allocated bit for this bad object. This "bad object isolation" ensures that the object will not be reused for future allocations, preserving the contents for Exterminator's subsequent use.

After every deallocation, DieFast checks both the preceding and the 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 objects will differ from run to run. Combined with the check performed for each allocation, this approach probabilistically ensures a full heap integrity check every $H/3$ object allocations and deallocations, where H is the number of objects on the heap.

3.4 Modes of Operation

Exterminator can be used in two modes of operation: an iterative mode suitable for testing or whenever all inputs are available, and a replicated mode that is suitable both for testing and for deployment. Both 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 a number of 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 gather enough information to generate runtime patches for an individual error, as we show in Section 6.2. When run with a correcting memory allocator that incorporates these changes (described in detail in Section 5.3), 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. The replicated mode of operation allows Exterminator to operate on deployed software.

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 equipped with a correcting allocator. This organization allows Exterminator to discover and fix errors.

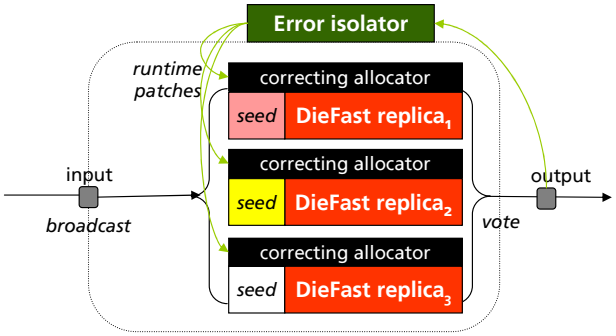

Figure 4. Exterminator's replicated architecture: replicas are equipped with different seeds that fully randomize their DieFast-based heaps (Section 3.3), input is broadcast to all replicas, and output goes to a voter. A crash, output divergence, or signal from DieFast triggers the error isolator (Section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

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

If DieFast signals an error, the replicas that dump a heap image do not have to stop executing. If their output continues to be in agreement, they can continue executing concurrently with the error isolation process. When the runtime patch generation process is complete, that process signals the running replicas to tell the correcting allocators to reload their runtime patches. Thus, subsequent allocations in the same process will be patched on-the-fly without interrupting execution.

4. Error Isolation

Exterminator executes its probabilistic error isolation algorithm on the multiple heap images that it generates either in its replicated or iterative modes. The algorithm operates by searching for discrepancies across these 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 generates no false positives for overflows, and limits the number of false positives for 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 to the same logical object across all heaps, then Exterminator considers it to be the same logical pointer, and thus not a discrepancy. We prove that this algorithm is unlikely to misidentify an integer as a pointer in the Appendix.

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 object's addresses (e.g., certain 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.

False negative analysis: 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 joint probability. □

False negative analysis: 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. It next scans the heap images for a matching **culprit**, the source of the overflow into a victim. We assume that overflows are deterministic, so the culprit will be the same distance δ bytes away from the victim in every heap image.

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 overflow list.

False positive analysis: 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 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 low scores, and large overflow strings present in many heap images get high scores.

After overflow processing completes, Exterminator generates a runtime patch for the top-ranked overflow for each 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.

Overwritten Dangling Object

When a freed object has been overwritten with identical values across all heap images, Exterminator assumes that a dangling pointer overwrite has occurred. As Theorem 1 shows, this situation is highly unlikely to occur for a buffer overflow. Exterminator then generates an appropriate repair patch, as Section 5.2 describes.

Intact Dangling Object

When a dangling pointer error leaves the target object intact, isolating the source of the error can be challenging. When DieFast probabilistically overwrites the dangling pointer with a canary, the program may crash if it dereferences this value. However, as Section 3.3 points out, a program may also treat the canary as data and propagate it through a computation. In this case, the error may not manifest for a long time, until the error finally propagates to output or causes the program to crash. Worse, every canary-filled freed object is a potential source of a dangling pointer error.

Exterminator's probabilistic canary filling resolves this problem. By filling freed objects with canaries with $p = 1/2$, Exterminator turns every execution into a Bernoulli trial. If overwriting a prematurely-freed object with canaries leads to an error, then its overwrite will correlate with a failed execution with probability $p > 1/2$. 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.

Exterminator counts the number of times h out of the total number of images n that the following predicate holds for each object: (canary-filled $\wedge$ failure) $\vee$ (not canary-filled $\wedge$ success). Exterminator currently rejects the null hypothesis (that the object is *not* a dangling pointer) only when its likelihood is less than one in 150,000. Assuming a normal distribution, this likelihood is approximately 4.358 standard deviations away from the mean:

$$\frac{2h-n}{\sqrt{n}} > 4.358$$

It thus takes a relatively large number of heap images (e.g., where the predicate holds 19/19 times) to identify the source of read-only dangling pointer errors.

Since it would be impractical to run with this number of replicas, Exterminator combines results from multiple runs. As objects do not map one-to-one across different executions, Exterminator identifies potentially dangled objects by their allocation and deallocation sites. For each of these pairs, Exterminator tracks the following statistics: the number of dynamic objects matching those sites, and the number of times the predicate held across all these objects.

Exterminator generates a repair patch for a particular allocation and deallocation site only when the above confidence level is reached.

5. Error Repair

We now describe how Exterminator uses the information from the error isolation algorithm to repair 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.

5.1 Buffer overflow repair

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

5.2 Dangling pointer repair

The repair patch for a dangling pointer consists of the combination of its allocation and deallocation site info, plus 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 deferral) by twice the distance between its premature free and the last allocation time, plus one: $2*(T-\tau)+1$.

This choice ensures that Exterminator will compute a correct patch in a logarithmic number of executions. As we show in Section 6.2, multiple iterations to correct pointer errors are rare in practice, because the last allocation time can be well past the time that the object should have been freed.

It is important to note that this deallocation deferral does not multiply its lifetime but rather its *drag* [37]. 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 just 21 allocations, increasing its lifetime by less than 1% (1021/1010). Section 6.3 empirically evaluates the impact of both overflow and dangling pointer repair on space consumption.

```
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 allocation pad (if any)
  // for this allocation 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 the site info for this pointer
  // (combined allocation and free sites).
  int allocS = getAllocSite (ptr);
  int freeS = computeFreeSite();
  setFreeSite (ptr, freeS);
  // Defer or free?
  int defer = deferralMap (allocS, freeS);
  if (defer == 0) {
    really_free (ptr);
  } else {
    deferralQ.push (ptr, clock + defer);
  }
}
```

Figure 5. Pseudo-code for the correcting memory allocator, which incorporates the runtime patches generated by the error isolator.

5.3 The Correcting Memory Allocator

The correcting memory allocator incorporates the runtime patches described above and applies them when appropriate. Figure 5 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 a tuple 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 to the object and the time to actually free the object (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.

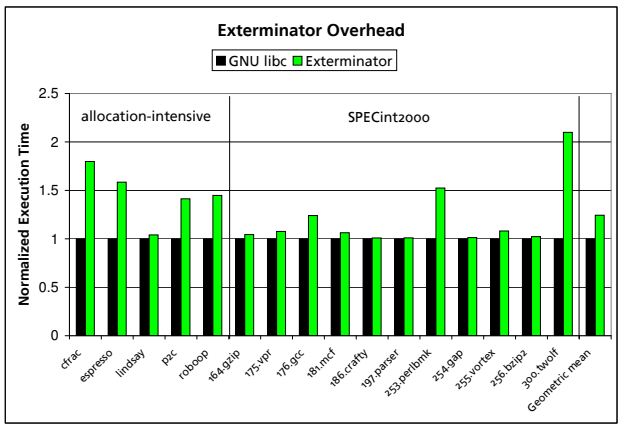

Figure 6. Runtime overhead for Exterminator across a suite of benchmarks, normalized to the performance of GNU libc (Linux) allocator.

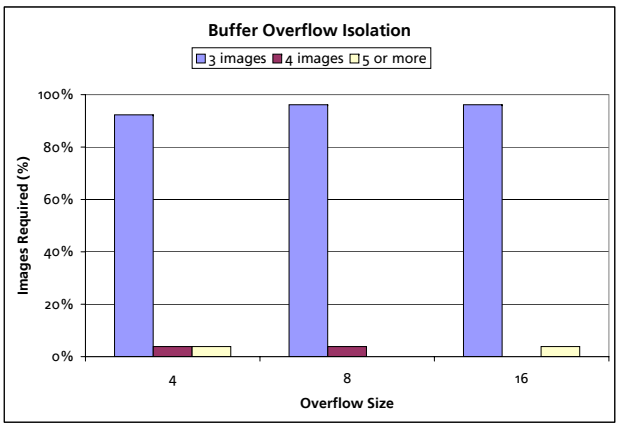

Figure 7. The number of images required for Exterminator to isolate and correct buffer overflows of different sizes.

5.4 Collaborative Repair

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 repair. 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/deallocation site pair. 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 and deallocation sites in a program, we expect these files to be compact and practical to transmit. For example, the uncompressed size of the runtime patches that Exterminator generates for injected errors in `espresso` was just 22K.

6. Results

Our evaluation answers the following questions:

1. What is the runtime overhead of using Exterminator without runtime patching?
2. How effective is Exterminator at finding and correcting memory errors, both for injected and real faults?
3. What is the space overhead of Exterminator's runtime patches?

6.1 Exterminator Runtime Overhead

We evaluate Exterminator's performance with the SPECint2000 suite [39] running reference workloads [3], 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, including the original DieHard paper [3, 18, 21], and because their high allocation-intensity stresses memory management performance.

For the execution time experiments, we fix Exterminator's heap size at 32 megabytes per size class (the same value used in the DieHard paper), except for `176.gcc` and `253.perlbmk`, which require 64 megabytes per size class. Since larger heap sizes worsen performance by degrading L2 and TLB locality, this choice makes our experimental execution time results conservative.

We use the smallest possible heap for our fault injection experiments with `espresso` (8 megabytes per size class), since smaller

[3] We are unable to run `252.eon` either with GNU libc or Exterminator.

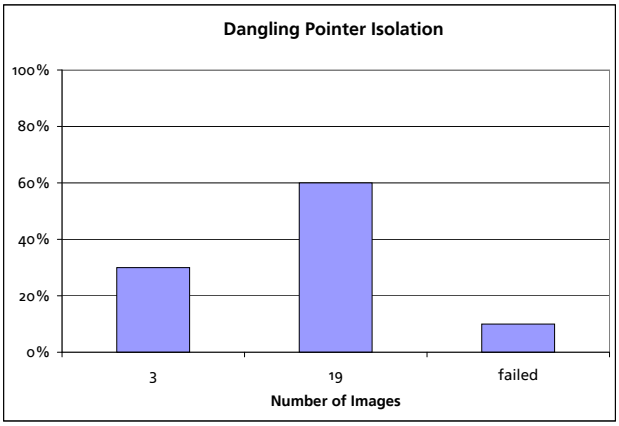

Figure 8. The number of images required for Exterminator to isolate and correct dangling pointer errors.

heap size multiples (values of M) stress Exterminator's ability to isolate errors.

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

We focus on the non-replicated case, which we expect to be a key limiting factor for Exterminator's performance as the number of available processing cores grows. This case assumes that additional processing cores are available to run replicas, and ignores the cost of voter-imposed synchronization.

We compare the runtime of Exterminator (DieFast plus the correcting allocator, without any patches) to the GNU libc allocator. This allocator is based on the Lea allocator [23], which is among the fastest available [4]. Figure 6 shows that, versus this allocator, Exterminator degrades performance by from 1% (`186.crafty`) to 109% (`300.twolf`), with a geometric mean of 24.3%. While Exterminator's overhead is substantial for the allocation-intensive suite (geometric mean: 43.4%), it is significantly less pronounced across the SPEC benchmarks (geometric mean: 16.5%).

6.2 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 both 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 iterated 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.

Figure 7 presents the result of triggering 26 different buffer overflows for three different sizes (4, 8, and 16 bytes) in the `espresso` benchmark. The number of images required to isolate and correct these errors is generally just 3, although occasionally there are outlier cases, where the number of images required reaches at most 5. 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 4% false negative rate we observe here.

Figure 8 presents the result of running 10 dangling pointer injection experiments. We have verified that in the three runs where three images were required, the program partially overwrote the contents of the canary-filled dangling pointer. Six of the injected faults are read-only dangling pointer errors, and the required number of images is consequently large, as the analysis in Section 4.2 shows.

One case is particularly notable because it violates our assumption that errors are deterministic. In this case, writing the canary into the dangled object triggers a cascade of errors that corrupts the heap. Worse, each random canary results in different corruption, so Exterminator cannot process it.

Real Faults

We also tested Exterminator with an actual bug in a real application, the Squid web caching server. Version 2.3s5 has a buffer overflow; certain inputs cause Squid to crash with either the GNU libc allocator or the Boehm-Demers-Weiser collector [3, 31].

We ran Squid three times under Exterminator 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.

6.3 Space Overhead

Exterminator's approach to correcting memory errors consumes additional space, either by padding allocations or by deferring deallocations. We measure the space overhead for buffer overflow repairs by multiplying the size of the pad by the total number of dynamic objects that Exterminator patches. For the buffer overflow experiment with overflows of size 16, this value is at most 224 bytes.

We measure space overhead for dangling pointer repairs 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.

7. Related Work

This section describes work most closely related to Exterminator.

7.1 Randomized Memory Managers

Several memory management systems employ some degree of randomization, including locating the heap at a random base address [5, 30], adding random padding to allocated objects [6], shuffling recently-freed objects [22], or a mix of padding and object deferral [31]. This level of randomization is insufficient for Exterminator, which requires full heap randomization.

As noted previously, Exterminator builds on DieHard, whose goal is to provide probabilistic memory safety; Section 3.1 provides an overview. Exterminator substantially modifies and extends DieHard's heap layout and allocation algorithms. It also applies a novel, probabilistic algorithm that, in addition to tolerating errors, identifies and repairs them.

7.2 Automatic Repair

We are aware of only one other system that repairs errors automatically: Demsky et al.'s *automatic data structure repair* [10, 11, 12]. Guided by a formal description of the program's data structures (specified manually or derived automatically by Daikon [17]), automatic data structure repair enforces data structure consistency specifications. Exterminator attacks a different problem, namely that of isolating and repairing memory errors, and is orthogonal and complementary to data structure repair.

7.3 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 [24, 25, 26]. It operates by injecting lightweight tests into the source code; the result of these tests, in the form of a bit vector, can be processed to generate likely sources of the errors. This statistical processing differs entirely from Exterminator's probabilistic error isolation algorithms, although Exterminator's treatment of read-only dangling pointer errors as Bernoulli trials is similar to the approach of Liu et al. [26]. Like statistical bug isolation, Exterminator can leverage the runs of deployed programs to improve its results. 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 repair them.

Delta debugging automates the process of identifying the smallest possible inputs that do and do not exhibit a given error [9, 27, 44]. 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 repair an error given just one erroneous input, regardless of its size.

7.4 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 [32] to software fault tolerance [33]. 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 [19]. Exterminator goes beyond this previous work by characterizing each memory error so specifically that a correction can be automatically generated for it.

7.5 Memory Managers

Conservative garbage collection can be used with unmodified C and C++ binaries to prevent dangling pointer errors [7], but it does not prevent buffer overflows. Exterminator's error isolation and correction is orthogonal to garbage collection, and could be combined with a randomized conservative collector to preclude dangling pointer errors, while detecting and correcting overflows.

Finally, there has been a long history of debugging memory allocators; the documentation for one of them, *mpatrol*, includes a list of over ninety such systems [36]. Notable recent allocators with debugging features include dnmalloc [43], Heap Server [22], and version 2.8 of the Lea allocator [23, 35]. These tools detect certain memory errors. Exterminator not only prevents some errors, but also pinpoints the location of and repairs buffer overflows and dangling pointer errors.

8. Conclusion

This paper presents Exterminator, a system that automatically corrects heap-based memory errors in C and C++ programs. Exterminator operates entirely at the runtime level on unaltered binaries, and consists of three key components: (1) DieFast, a probabilistic debugging allocator based on DieHard [3] that exposes errors instead of masking them, (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 repair memory errors. Exterminator is not only suitable for use during testing, but also can automatically repair deployed programs.

8.1 Future Work

While Exterminator can effectively locate and repair memory errors on the heap, it does not yet address stack errors. We are actively investigating the use of binary instrumentation to apply similar techniques to the stack.

Exterminator currently requires programs to be deterministic in their allocation patterns and their storage to heap objects. Individual executions of a program are expected to yield the same sequence of object allocation identifiers (i.e., object *i* always refers to the same semantic object).

These requirements make Exterminator well-suited for serial programs, but less useful for multi-threaded programs. While Exterminator can run programs with a small amount of non-determinism, its isolation algorithm is not robust with respect to disruptions to the allocation sequence.

While simulation or deterministic replay mechanisms [8] would enable Exterminator to work reliably for multi-threaded programs, we plan to explore a variety of ways to directly handle programs with extensive non-determinism. One way to improve its behavior in the face of non-determinism would be for Exterminator to associate object identifiers with thread ids. This change would prevent multiple threads from disrupting a global allocation sequence. We are especially interested in combining statistical alignment techniques from machine learning with information gleaned from locks and other synchronization variables to let us recover allocation sequences across multiple threads.

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] 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.

[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 (PLDI)*, pages 158–168, New York, NY, USA, 2006. ACM Press.

[4] 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.

[5] 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.

[6] 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.

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

[8] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, Aug. 1998.

[9] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 342–351, 2005.

[10] 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 *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 233–244, New York, NY, USA, 2006. ACM Press.

[11] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA '03: 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.

[12] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 176–185, 2005.

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

[14] 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.

[15] D. Dhurjati, S. Kowshik, and V. Adve. Safecode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 144–157, New York, NY, USA, 2006. ACM Press.

[16] 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.

[17] 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 (ICSE'00)*, pages 449–458, New York, NY, USA, 2000. ACM Press.

[18] D. Grunwald, B. Zorn, and R. Henderson. Improving the cache local-

ity 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.

[19] Z. Guo, G. Jiang, H. Chen, and K. Yoshihira. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. *dsn*, 0:259–268, 2006.

[20] 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.

[21] 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.

[22] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic. Comprehensively and efficiently protecting the heap. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 207–218, New York, NY, USA, 2006. ACM Press.

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

[24] 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 (PLDI 2003)*, 2003.

[25] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26, New York, NY, USA, 2005. ACM Press.

[26] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *ESEC/FSE-13: 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.

[27] G. Misherghi and Z. Su. Hdd: hierarchical delta debugging. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 142–151, New York, NY, USA, 2006. ACM Press.

[28] 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.

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

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

[31] 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.

[32] 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.

[33] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software Implemented Fault Tolerance. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization*, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[34] 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.

[35] 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.

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

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

[38] 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.

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

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

[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] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *ESEC/FSE-7: 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.

9. Appendix

9.1 Probabilistic pointer disambiguation

To disambiguate pointers from data, Exterminator combines standard approaches from conservative garbage collection with probabilistic techniques. It first eliminates from consideration any value that, if interpreted as an address, would point outside the heap area.

After performing these tests, Exterminator relies on its randomized heaps to identify pointers with high probability. Exterminator checks whether these addresses point to the same offset in equivalent objects (same object id) across all heap images. If they do, then Exterminator considers the value to be a valid pointer. Exterminator also handles the case where pointers point into dynamic libraries, which newer versions of Linux place at random base addresses.

The following formula gives the precision of probabilistic pointer disambiguation.

Theorem 4. *Let H be the number of objects on the heap, and k the number of replicas. Then the false positive rate (misidentifying an integer as a pointer) is at most:*

$$P(\text{pointer id false positive}) \quad \leq \quad \frac{1}{H^{k-1}}.$$

Proof. Consider the case where an object has a field that takes on the values of any address on the heap uniformly at random (rather than taking on any possible value; this is the worst case). For the first replica, this value points to some object i. The likelihood of a random value pointing to the same object in a particular heap image is $1/H$, so across k heaps, the odds of a random value pointing to the same object are at most $(1/H)^{k-1}$. □

Probabilistic pointer disambiguation provides excellent odds of success. As a concrete example, if there are one million slots on the heap and three heap images, the odds of misidentifying a random integer as a pointer will be at most one in a trillion.