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.

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. 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 [7] and Valgrind [10], which may 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. This process can be costly and time-consuming. For example, 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 [17].

Because memory errors are so difficult to find and fix, researchers have proposed many solutions that fall roughly into two categories: detection, which prevents errors from being exploited and potentially allows them to be debugged more easily; and toleration, where the effects of errors are mitigated. *Fail-stop* systems are compiler-based approaches that may require access to source code, and abort programs when they perform illegal operations like buffer overflows [1, 2, 6, 9].

Fault tolerant runtime systems, which attempt to hide the effect of errors, have also been proposed. Rinard's *failure-oblivious* systems are also compiler-based, but manufacture read values and drop or cache illegal writes for later reuse [13, 14]. The Rx system [12] uses logging and replay, with potential perturbation, to provide fault tolerance. Our previous work, DieHard [3, 4], uses heap over-provisioning, layout randomization, and optional voting-based replication to reduce the likelihood that an error will have any effect (see Section 3.1 for an overview). DieHard provides *probabilistic memory safety*, giving the application the illusion of having an infinite heap with a well-defined probability.

Contributions: This paper presents **Exterminator**, a runtime system that not only tolerates but also detects, isolates, and corrects two important classes of heap-based memory errors with high probability. 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, 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 processes one or more heap images to try to locate the source of the error. This error isolation algorithm has provably low false positive and false negative rates. It can distinguish buffer overflows and dangling pointer errors because they tend to produce distinct patterns of heap corruption.

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.

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 (around 25%), Exterminator effectively isolates and corrects both injected and real memory errors, including buffer overflows in the Squid web cache server and the Mozilla web browser.

2. MEMORY ERRORS

Incorrect programs exhibit a variety of errors related to heap objects, including *dangling pointers*, where a heap object is freed while it is still live; *invalid frees*, where a program deallocates an object that was never returned by the allocator; *double frees*, where a heap object is deallocated multiple times without an intervening allocation; *uninitialized reads*, where the program, despite using all pointers correctly, reads memory that has never been initialized; and *out-of-bound writes*, where the memory address to be written is computed by using a valid pointer to an object but an incorrect offset or index, so that the address computed lies outside the object. We use the term *buffer overflow* to refer to an out-of-bound write whose offset from a base pointer is positive and too large. (Out-of-bound writes where the offset is negative appear to be rather less common in practice.).

Errors such as double frees and invalid frees, if not properly handled, can result in inconsistent allocator metadata and are a potential security vulnerability. These errors can lead to heap corruption or abrupt program termination. Out-of-bound writes and dangling pointers may result in corruption of either allocator metadata or application objects. Uninitialized reads, because the values read are not specified by the language semantics, can affect application execution in arbitary ways. Because good allocator design can mitigate the effect of double frees and invalid frees, buffer overruns and dangling pointer errors are currently the most commonly exploited heap errors, and hence the most important to address.

While DieHard probabilistically tolerates dangling pointers and buffer overflows of heap objects, Exterminator both detects and permanently corrects them. Exterminator's allocator (DieFast) shares DieHard's immunity from double frees and invalid frees. Exterminator does not currently address uninitialized reads, reads outside the bounds of an object, or out-of-bound writes with negative offsets.

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.

3.1 DieHard Overview

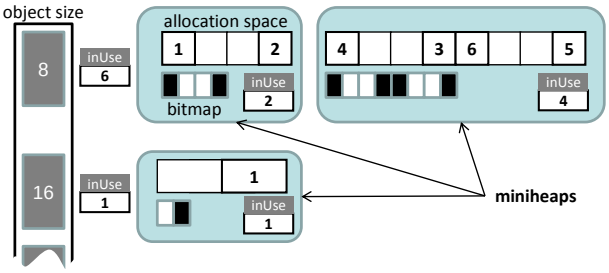

Figure 1: The adaptive (new) DieHard heap layout, used by Exterminator. Objects in the same size class are allocated randomly from separate *miniheaps*, which combined hold M times more memory than required (here, $M = 2$).

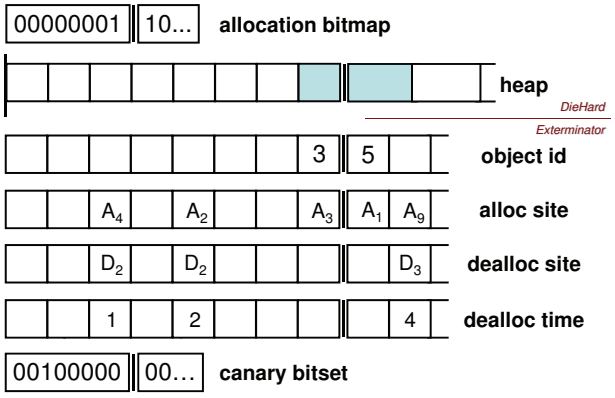

Figure 2: An abstract view of Exterminator's heap layout. Metadata below the horizontal line contains information used for error isolation and correction (see Section 3.2).

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 to be M times larger than the maximum needed by the application [4] (see Figure 1). This version of DieHard allocates memory from increasingly large chunks that we call *miniheaps*. Each miniheap contains objects of exactly one size. DieHard allocates new miniheaps to ensure that, for each size, the ratio of allocated objects to total objects is never more than $1/M$. Each new miniheap is twice as large, and thus holds twice as many objects, as the previous largest miniheap.

Allocation randomly probes a miniheap's bitmap for the given size class for a 0 bit, indicating a free object available for reclamation, and sets it to 1. This operation takes $O(1)$ expected time. Freeing a valid object resets the appropriate bit, which is also a constant-time operation. 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.

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 inde-

pendent. Replication thus exponentially decreases the likelihood of a memory error affecting output, since the probability of an error corrupting a majority of the replicas is low.

3.2 Exterminator's Heap Layout

Figure 2 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 bit**.

An object id of n means that the object is the nth object allocated. Exterminator uses object ids to identify objects across heaps in multiple program executions. These ids are needed because object addresses cannot be used to identify them across differently-randomized heaps. The site information fields capture the calling context for allocations and deallocations: a 32-bit hash of the least significant bytes of the five most-recent return addresses. The canary bit indicates if the object was filled with canaries (see Section 3.3). All of this metadata is initialized when an object is allocated and persists after the object is freed until a new object is allocated in its place.

The space overhead of this out-of-band metadata plus the allocation bit is 16 bytes plus two bits of space overhead per object. This amount is comparable to that of typical freelist-based memory managers like the Lea allocator, which prepend an 8-byte or 16-byte header (on 32-bit or 64-bit systems) to each object [8].

3.3 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. 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 out-of-bound writes that are just beyond the start or end of an object 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 acts as the one preceding the next object. Second, because allocated objects are separated by (on average) $M-1$ freed objects on the heap, we use freed space to act as fence-posts.

Random Canaries

Traditional debugging canaries include values, such as the hexadecimal value `0xDEADBEEF`, that are readily distinguished from normal program data in a debugging session. 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 value and heap addresses are random and differ on every execution, any fixed data value (likewise, any given pointer) has a low probability of colliding with the canary; this ensures a low false positive rate (see Theorem 2). To increase the likelihood of detecting an error, DieFast always sets the last bit of the canary value to 1. Setting this bit will cause an alignment error if the canary is dereferenced, but still 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 value as a pointer will likely trigger a segmentation violation or alignment error.

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 may affect the correctness of the computation.

When reading from a canary-filled dangled object causes a program to run awry, it can become difficult to isolate the error. In the worst case, half of the heap could be filled with freed objects, all overwritten with canary values. All of these objects would then be potential sources of dangling pointer errors.

In cumulative mode, Exterminator prevents this scenario by making a random choice every time an object is freed; rather than always filling the freed object with canaries and setting the associated canary bit, it performs this filling and bit-setting action with probability p. 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 it is supposed to contain (as indicated by the canary bitset) 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. By checking canary integrity on each allocation, DieHard can be expected to detect heap corruption within approximately 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 both the subsequent and the preceeding objects on each free allows DieFast to perform an inexpensive check for any out-of-bound writes, including "strided" object writes (e.g., `a[i + 32]`) that might jump over a subsequent object.

3.4 Modes of Operation

Exterminator can be used in three modes of operation: an iterative mode suitable for testing or whenever all program inputs can be made available for repeated execution, 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 (that is, 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.

As Figure 3 shows, Exterminator (like DieHard) 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. Exterminator also dumps heap images if any replica crashes because of a segmentation fault.

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. Once the runtime patch generation process has completed, it signals the running replicas 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

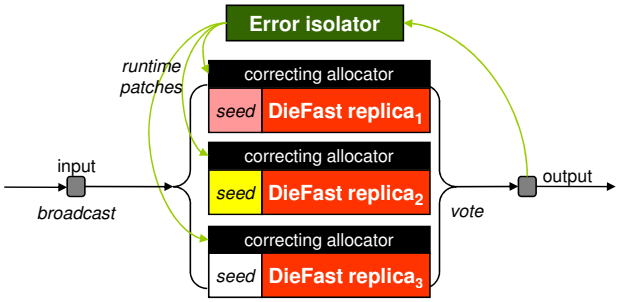

Figure 3: Exterminator's replicated architecture (Section 3.4). Replicas are equipped with different seeds that fully randomize their DieFast-based heaps (Section 3.3), input is broadcast to all replicas, and output goes to a voter. A crash, output divergence, or signal from DieFast triggers the error 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.

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 stored in freed objects to indicate the presence of an error. A corrupted canary (one that has been overwritten) can mean two things. If the same object (identified by object id) across all heap images has the same corruption, then the error is likely to be a dangling pointer. If canaries are corrupted in multiple freed objects, then the error 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, Exterminator considers it to be a candidate **victim** of an overflow.

To identify victim objects, Exterminator compares the contents of equivalent objects, as identified by their object id across all heaps. 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 occurrences 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 pseudorandom 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 all the heaps, it considers it to be legitimately different, and not an indication of a buffer overflow.

For small 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 an object on all k heaps is:*

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

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

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

THEOREM 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 $(1/256)^b$. □

4.2 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 object that is likely to be the source of the overflow into a victim. Because Exterminator assumes that overflows are deterministic when operating in iterative or replicated mode, 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. Note that, while Exterminator only considers positive values of δ, these values may be arbitrarily large.

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.

4.3 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

Unlike iterative and replicated mode, cumulative mode focuses on detecting, isolating, and correcting errors that happen in the field.

In this context, replication, identical inputs, and deterministic execution are infeasible. Worse, program errors may manifest themselves in ways that are inherently hard to detect. For example, a program that reads a canary written into a free object may fail immediately, or may execute incorrectly for some time.

Our approach to error detection in this mode is to consider exceptional program events, such as premature termination, raising unexpected signals, etc., to be evidence that memory was corrupted during execution. We counter the lack of error reproducibility in these cases with statistical accumulation of evidence before assuming an error needs to be corrected. Exterminator isolates memory errors in cumulative mode by computing summary information accumulated over multiple executions, rather than by operating over multiple heap images.

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.

Exterminator's randomized allocator allows us to compute the probability of certain properties in the heap. For example, the probability of an object occurring on a given miniheap can be estimated given the miniheap size and the number of miniheaps. If objects from some allocation site are sources of overflows, then those objects will occur on miniheaps containing corruptions more often than expected. Exterminator tracks how often objects from each allocation site occur on corrupted miniheaps across multiple runs. Using this information, it uses a statistical hypothesis test that identifies sites that occur with corruption too often to be random chance, and identifies them as overflow culprits (see [11] for more details).

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

5.2 Dangling Pointer Isolation

As with buffer overflows, Exterminator's dangling pointer isolator computes summary information over multiple 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. In this scenario, if the program reads canary data through the dangling pointer, the program may crash. Thus writing the canary for that object increases the probability that the program will later crash. Conversely, if an object is not freed prematurely, then overwriting it with canaries has no influence on the failure or success of the program. Exterminator then uses the same hypothesis testing framework as its buffer overflow algorithm to identify sources of dangling pointer errors.

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 (those that allocate large numbers of objects) 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.

Exterminator's ability to correct memory errors has several inherent limitations. Exterminator can only correct finite overflows, because it tries to contain any given overflow by finite over-allocation. Similarly, Exterminator corrects dangling pointer errors by inserting finite delays before freeing particular objects. Finally, Exterminator cannot correct memory errors when the evidence it uses to locate these errors is destroyed, such as when an overflow overwrites most of the heap, or when a program with a dangling pointer error runs long enough to re-allocate the dangled object.

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 amount of padding needed to contain the overflow (δ + the size of the overflow string). 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 hash and an amount of 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 allocation time at which the program crashed or Exterminator detected heap corruption. Exterminator has no way of knowing how long the object is supposed to live, so computing an exact delay is impossible. Instead, it extends the object's lifetime (delays its freeing) by twice the distance between its premature freeing and the time of crashing or detection, plus one: $2 \times (T - \tau) + 1$.

It is important to note that this deallocation deferral does not multiply object lifetimes but rather their *drag* [15]. 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 correct lifetime (1010 allocations) by less than 1% (1021/1010).

6.3 The Correcting Memory Allocator

The correcting memory allocator incorporates the runtime patches described above and applies them when appropriate.

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.

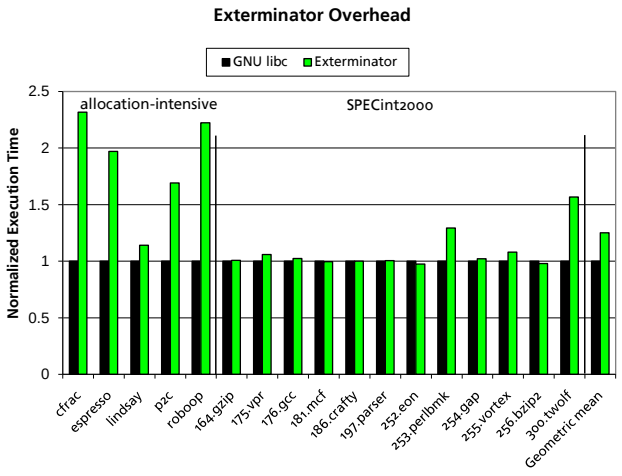

Figure 4: Runtime overhead for Exterminator across a suite of benchmarks, normalized to the performance of GNU libc (Linux) allocator.

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 checks the deferral queue on every allocation to see if any object should now be freed. It then checks whether the current allocation site has an associated pad size. If so, it adds the pad size to the allocation request, and forwards the allocation request to the underlying 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 pad size required for any allocation site, and the maximal deferral amount for any given allocation site / 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 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 (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?

7.1 Exterminator Runtime Overhead

We evaluate Exterminator's performance with the SPECint2000 suite [16] 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 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 [8], which is among the fastest available [5]. Figure 4 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%), for which 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.

Note that Exterminator's approach to correcting memory errors does not impose additional execution time overhead in the presence of patches. However, it can consume additional space, either by padding allocations or by deferring deallocations.

Buffer overflows: We triggered 10 different buffer overflows each of three different sizes (4, 20, and 36 bytes) by intentionally undersizing objects in the `espresso` benchmark. In every case, three images were required to isolate and correct these errors. 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), but we observed a 0% false negative rate. The most space overhead we observe is a total increase of 2816 bytes.

Dangling pointer errors: We then triggered 10 dangling pointer faults in `espresso` with Exterminator running in iterative and in cumulative modes. Recall that in iterative mode, Exterminator always fills freed objects with canaries, while it does so probabilistically when running in cumulative mode (see Section 3.3).

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 that Exterminator requires to isolate the error.

However, in cumulative mode, probabilistic canary-filling enables Exterminator to isolate all injected errors, including the read-only dangling pointer errors. For runs where no large-scale heap corruption occurs, Exterminator requires between 22 and 30 execu-

tions 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.

The space overhead of the derived runtime patches 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.

Real Faults

We also tested Exterminator with actual bugs in two applications: the Squid web cache 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.

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.

8. 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 not only is suitable for use during testing, but also can automatically correct deployed programs.

9. ACKNOWLEDGMENTS

We thank Sam Guyer, Mike Hicks, Erik Learned-Miller, Sarah Osentoski, Martin Rinard, and Guy Steele 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.

10. 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 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301. ACM Press, June 1994.

[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. ACM Press, May 2005.

[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. ACM Press, June 2006.

[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*, pages 114–124. ACM Press, June 2001.

[6] 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. ACM Press, June 2006.

[7] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138. USENIX, Jan. 1992.

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

[9] 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. ACM Press, Jan. 2002.

[10] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100. ACM Press, June 2007.

[11] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11. ACM Press, June 2007.

[12] 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*, pages 235–248. ACM Press, Oct. 2005.

[13] 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 20th Annual Computer Security Applications Conference*, pages 82–90. IEEE Computer Society, Dec. 2004.

[14] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Sixth Symposium on Operating Systems Design and Implementation*, pages 303–316. USENIX, Dec. 2004.

[15] N. Röjemo and C. Runciman. Lag, drag, void, and use: heap profiling and space-efficient compilation revisited. In *Proceedings of First International Conference on Functional Programming*, pages 34–41. ACM Press, May 1996.

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

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