Exterminator

Automatically Correcting Memory Errors with High Probability

www.vtucs.com

CONTENTS

1. Introduction ---
2. Memory errors --
3. Software Architecture ---
    3.1. DieHard Overview --
    3.2. Exterminator's Heap Layout ----------------------------------
    3.3. DieFast: A Probabilistic Debugging Allocator --------------------------
    3.4. Modes of Operation --
4. Error Isolation (ITERATIVE AND REPLICATED) ---------------------------
    4.1. Buffer overflow detection ---------------------------------------
    4.2. Dangling Pointer Isolation -------------------------------------
5. CUMULATIVE ERROR ISOLATION ---
    5.1. Buffer overflow detection ---------------------------------------
    5.2. Dangling pointer isolation --------------------------------------
6. Error Correction --
    6.1. Buffer overflow correction -------------------------------------
    6.2. Dangling pointer correction ------------------------------------
    6.3. The Correcting Memory Allocator ---------------------------------
    6.4. Collaborative Correction --
7. Results ---
    7.1. Exterminator Runtime Overhead ----------------------------------
    7.2. Memory Error Correction ---
    7.3 Patch Overhead --
8. Related Work ---
    8.1. Randomized Memory Managers ------------------------------------
    8.2. Automatic Repair --
    8.3. Automatic Debugging ---
    8.4. Fault Tolerance ---
    8.5. Memory Managers ---

www.vtucs.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.

www.vtucs.com

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 de allocates 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 and Valgrind , 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.

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

www.vtucs.com

Contributions: This paper presents Exterminator, a runtime system that not only tolerates but also detects and corrects heap-based memory errors. Exterminator requires neither source code nor programmer intervention, and fixes existing errors without introducing new ones. To our knowledge, this system is the first of its kind. Exterminator relies on an efficient probabilistic debugging allocator that we call **DieFast**. DieFast is based on DieHard's allocator, which ensures that heaps are independently randomized. However, while DieHard can only probabilistically tolerate errors, DieFast probabilistically detects them.

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

Once Exterminator locates a buffer overflow, it determines the allocation site of the overflowed object, and the size of the over flow. For dangling pointer errors, Exterminator determines both the allocation and deletion sites of the dangled object, and computes how prematurely the object was freed.

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

www.vtucs.com

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

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

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

www.vtucs.com

2. Memory Errors

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

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

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

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

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

- **Limitations**

Exterminator's ability to correct both dangling pointer errors and buffer overflows has several limitations. First, Exterminator assumes that buffer overflows always corrupt memory at higher addresses—that is, they are forward overflows. While it is possible to extend Exterminator to handle backwards overflows, we have not implemented this

www.vu...com

functionality. Exterminator can only correct finite overflows, so that it can contain any given overflow by overallocation. Similarly, Exterminator corrects dangling pointer errors by inserting finite delays before freeing particular objects. Finally, in iterated and replicated modes, Exterminator assumes that overflows and dangling pointer errors are

deterministic. However, the cumulative mode does not require deterministic errors.

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

www.vtucs.com

3. Software Architecture

Exterminator's software architecture extends and modifies DieHard to enable its error isolating and correcting properties.

3.1 DieHard Overview

The DieHard system includes a bitmap-based, fully-randomized memory allocator that provides probabilistic memory safety. The latest version of DieHard, upon which Exterminator is based, adaptively sizes its heap be M times larger than the maximum needed by the application (see Figure 2). This version of Die-Hard allocates memory from increasingly large chunks that we call miniheaps. Each miniheap contains objects of exactly one size. If an allocation would cause the total number of objects to exceed1/M, DieHard allocates a new miniheap that is twice as large as the previous largest miniheap.

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

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

DieHard optionally uses replication to increase the probability of successful execution. In this mode, it broadcasts inputs to a number of replicas of the application process, each equipped with a different random seed. A voter intercepts and compares outputs across the replicas, and only actually generates output agreed on by a plurality of the replicas.

www.wlcs.com

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.

www.vtucs.com

3.2 Exterminator's Heap Layout

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

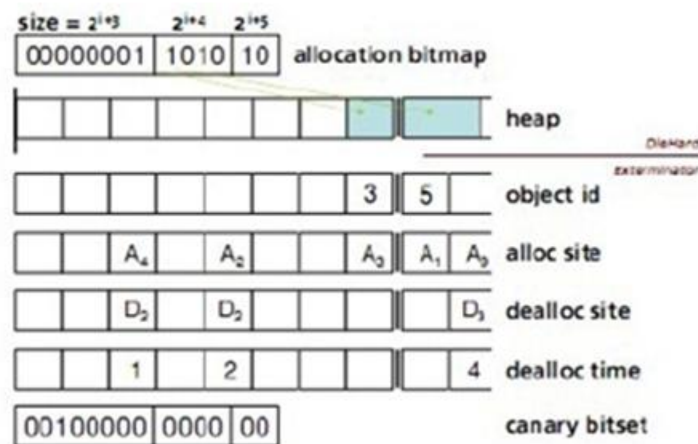


Figure 1. An abstract view of Exterminator's heap layout. Metadata below the horizontal line contains information used for error isolation and correction (see Section 3.2).

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

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

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

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 free list-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.

www.vtucs.com

3.3 DieFast: A Probabilistic Debugging Allocator

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

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

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

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

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-

www. .com

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 lay out is header less, one fence-post serves double duty: a fence-post following an object can act as the one preceding the next object. Second, because allocated objects are separated by E(M−1) freed objects on the heap, we use freed space to act as fence-posts.

Random Canaries

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

DieFast instead uses a random 32-bit value set at startup. Sinceboth the canary and heap addresses are random and differ on every execution; any fixed data value has a low probability of colliding with the canary, thus ensuring a low false positive rate (see Theorem 2). To increase the likelihood of detecting an error, DieFast sets the last bit of the canary. Setting this bit will cause an alignment error if the canary is dereferenced, but keeps the probability of an accidental collision with the canary low (1/231).

Probabilistic Fence-posts

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

www.vtucs.com

segmentation violation.

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

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

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

Probabilistic Error Detection

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

After every deallocation, DieFast checks both the preceding and subsequent objects. For each of these, DieFast checks if they are free. If so, it performs the same canary check

as above. Recall that because DieFast's allocation is random, the identity of these adjacent objects will differ from run to run. Checking the predecessor and successor on each free allows DieFast to detect buffer overruns immediately upon object deallocation.

3.4 Modes of Operation

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

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

www.vtucs.com

- ***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 suffice for Exterminator to generate runtime patches for an individual error, as we show in Section 7.2. When run with a correcting memory allocator that incorporates these changes (described in detail in Section 6.3), these patches automatically fix the isolated errors.

- ***Replicated Mode***

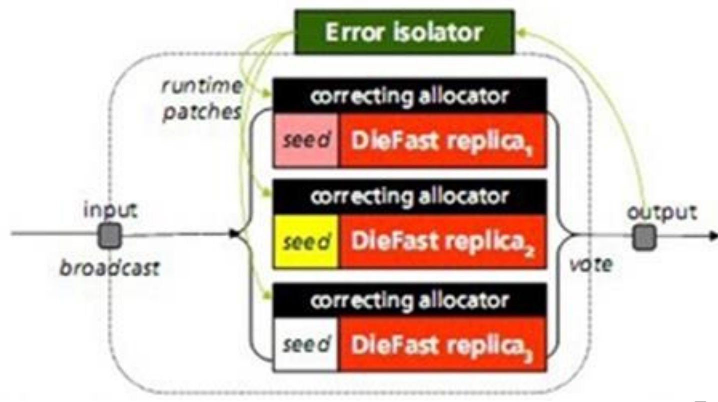
The iterated mode described above works well when all inputs are available so that re-running an execution is feasible. However, when applications are deployed in the field, such inputs may not be available, and replaying may be impractical. The replicated mode of operation allows Exterminator to correct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

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

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

If DieFast signals an error, the replicas that dump a heap image do not have to stop executing. If their output continues to be in agreement, they can continue executing concurrently with the error isolation process. When the runtime patch generation process is 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 onthe-fly without interrupting execution.

www.vtucs.com



www.vtucs.com

- ***Cumulative Mode***

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

When operating in cumulative mode, Exterminator reasons about objects grouped by allocation and de-allocation 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.

www.vtucs.com