Tolerating and Correcting Memory Errors in C and C++

Ben Zorn
Microsoft Research

In collaboration with:
Emery Berger and Gene Novark, UMass - Amherst
Karthik Pattabiraman, UIUC
Vinod Grover and Ted Hart, Microsoft Research
Focus on Heap Memory Errors

- Buffer overflow

```c
char *c = malloc(100);
c[100] = 'a';
```

- Dangling reference

```c
char *p1 = malloc(100);
char *p2 = p1;
free(p1);
p2[0] = 'x';
```
Approaches to Memory Corruptions

- Rewrite in a safe language
- Static analysis / safe subset of C or C++
  - SAFECODE [Adve], PREfix, SAL, etc.
- Runtime detection, fail fast
  - Jones & Lin, CRED [Lam], CCured [Necula], etc.

- Tolerate Corruption and Continue
  - Failure oblivious [Rinard] (unsound)
  - Rx, Boundless Memory Blocks, ECC memory
  - DieHard / Exterminator, Samurai
Fault Tolerance and Platforms

- Platforms necessary in computing ecosystem
  - Extensible frameworks provide lattice for 3rd parties
  - Tremendously successful business model
  - Examples: Window, iPod, browser, etc.

- Platform power derives from extensibility
  - Tension between isolation for fault tolerance, integration for functionality
  - **Platform only as reliable as weakest plug-in**
  - Tolerating bad plug-ins necessary by design
Research Vision

- Increase robustness of installed code base
  - Potentially improve millions of lines of code
  - Minimize effort – ideally no source mods, no recompilation

- Reduce requirement to patch
  - Patches are expensive (detect, write, deploy)
  - Patches may introduce new errors

- Enable trading resources for robustness
  - E.g., more memory implies higher reliability
Outline

- Motivation
- Exterminator
  - Collaboration with Emery Berger, Gene Novark
  - Automatically corrects memory errors
  - Suitable for large scale deployment
- Critical Memory / Samurai
  - Collaboration with Karthik Pattabiraman, Vinod Grover
  - New memory semantics
  - Source changes to explicitly identify and protect critical data
- Conclusion
DieHard Allocator in a Nutshell

- With Emery Berger (PLDI’06)
- Existing heaps are packed tightly to minimize space
  - Tight packing increases likelihood of corruption
  - Predictable layout is easier for attacker to exploit
- Randomize and over provision the heap
  - Expansion factor determines how much empty space
  - Does not change semantics
- Replication increases benefits
- Enables analytic reasoning

Normal Heap

DieHard Heap
DieHard in Practice

- **DieHard (non-replicated)**
  - Windows, Linux version implemented by Emery Berger
  - Try it right now! ([http://www.diehard-software.org/](http://www.diehard-software.org/))
  - Adaptive, automatically sizes heap
  - Mechanism automatically redirects malloc calls to DieHard DLL

- **Application: Firefox & Mozilla**
  - Known buffer in version 1.7.3 overflow crashes browser

- **Experience**
  - Usable in practice – no perceived slowdown
  - Roughly doubles memory consumption with 2x expansion
    - FireFox: 20.3 Mbytes vs. 44.3 Mbytes with DieHard
DieHard Caveats

- Primary focus is on protecting heap
  - Techniques applicable to stack data, but requires recompilation and format changes

- Trades space, processors for memory safety
  - Not applicable to applications with large footprint
  - Applicability to server apps likely to increase

- In replicated mode, DieHard requires determinism
  - Replicas see same input, shared state, etc.

- DieHard is a brute force approach
  - Improvements possible (efficiency, safety, coverage, etc.)
Exterminator Motivation

- DieHard limitations
  - Tolerates errors probabilistically, doesn’t fix them
  - Memory and CPU overhead
  - Provides no information about source of errors

- “Ideal” solution addresses the limitations
  - Program automatically detects and fixes memory errors
  - Corrected program has no memory, CPU overhead
  - Sources of errors are pinpointed, easier for human to fix

- Exterminator = correcting allocator
  - Joint work with Emery Berger, Gene Novark
  - Plan: isolate / patch bugs while tolerating them
Exterminator Components

- Architecture of Exterminator dictated by solving specific problems
- How to detect heap corruptions effectively?
  - DieFast allocator
- How to isolate the cause of a heap corruption precisely?
  - Heap differencing algorithms
- How to automatically fix buggy C code without breaking it?
  - Correcting allocator + hot allocator patches
DieFast Allocator

- Randomized, over-provisioned heap
  - Canary = random bit pattern fixed at startup
  - Leverage extra free space by inserting canaries

- Inserting canaries
  - Initialization – all cells have canaries
  - On allocation – no new canaries
  - On free – put canary in the freed object with prob. P

- Checking canaries
  - On allocation – check cell returned
  - On free – check adjacent cells
Installing and Checking Canaries

Initially, heap full of canaries

Allocate

Free

Install canaries with probability $P$

Check canary

Allocate

Check canary
Heap Differencing

Strategy

- Run program multiple times with different randomized heaps
- If detect canary corruption, dump contents of heap
- Identify objects across runs using allocation order

Insight: Relation between corruption and object causing corruption is invariant across heaps

- Detect invariant across random heaps
- More heaps \(\Rightarrow\) higher confidence of invariant
Attributing Buffer Overflows

Ben Zorn, Microsoft Research

Tolerating and Correcting Memory Errors in C and C++

Precision increases exponentially with number of runs
Detecting Dangling Pointers (2 cases)

- Dangling pointer read/written (easy)
  - Invariant = canary in freed object X has same corruption in all runs

- Dangling pointer only read (harder)
  - Sketch of approach (paper explains details)
    - Only fill freed object X with canary with probability P
    - Requires multiple trials: \( \approx \log_2(\text{number of callsites}) \)
    - Look for correlations, i.e., X filled with canary => crash
    - Establish conditional probabilities
      - Have: \( P(\text{callsite X filled with canary | program crashes}) \)
      - Need: \( P(\text{crash | filled with canary}) \), guess “prior” to compute
Correcting Allocator

- Group objects by allocation site
- Patch object groups at allocate/free time
- Associate patches with group
  - Buffer overrun => add padding to size request
    - malloc(32) becomes malloc(32 + delta)
  - Dangling pointer => defer free
    - free(p) becomes defer_free(p, delta_allocations)
  - Fixes preserve semantics, no new bugs created

Correcting allocation may != DieFast or DieHard

- Correction allocator can be space, CPU efficient
- “Patches” created separately, installed on-the-fly
Deploying Exterminator

- Exterminator can be deployed in different modes
- Iterative – suitable for test environment
  - Different random heaps, identical inputs
  - Complements automatic methods that cause crashes
- Replicated mode
  - Suitable in a multi/many core environment
  - Like DieHard replication, except auto-corrects, hot patches
- Cumulative mode – partial or complete deployment
  - Aggregates results across different inputs
  - Enables automatic root cause analysis from Watson dumps
  - Suitable for wide deployment, perfect for beta release
  - Likely to catch many bugs not seen in testing lab
DieFast Overhead

Tolerating and Correcting Memory Errors in C and C++

Ben Zorn, Microsoft Research
Exterminator Effectiveness

- Squid web cache buffer overflow
  - Crashes glibc 2.8.0 malloc
  - 3 runs sufficient to isolate 6-byte overflow

- Mozilla 1.7.3 buffer overflow (recall demo)
  - Testing scenario - repeated load of buggy page
    - 23 runs to isolate overflow
  - Deployed scenario – bug happens in middle of different browsing sessions
    - 34 runs to isolate overflow
Outline

- Motivation
- Exterminator
  - Collaboration with Emery Berger, Gene Novark
  - Automatically corrects memory errors
  - Suitable for large scale deployment
- Critical Memory / Samurai
  - Collaboration with Karthik Pattabiraman, Vinod Grover
  - New memory semantics
  - Source changes to explicitly identify and protect critical data
- Conclusion
The Problem: A Dangerous Mix

Danger 1:
Flat, uniform address space

Danger 2:
Unsafe programming languages

Danger 3:
Unrestricted 3rd party code

Result: corrupt data, crashes, security risks
Critical Memory

- **Approach**
  - Identify **critical program data**
  - Protect it with **isolation & replication**

- **Goals:**
  - **Harden** programs from both SW and HW errors
    - Unify existing ad hoc solutions
  - Enable **local reasoning** about memory state
    - Leverage powerful static analysis tools
  - Allow **selective, incremental hardening** of apps
  - Provide **compatibility** with existing libraries, apps
Critical Memory: Idea

- Identify and mark some data as "critical"
  - Type specifier like `const`
- Shadow critical data in parallel address space (critical memory)
- New operations on critical data
  - `cload` – read
  - `cstore` - write

```c
int balance;

balance += 100;
if (balance < 0) {
    chargeCredit();
} else {
    // use x, y, etc.
}
```
Critical Memory: Example

```c
int buffer[10];
critical int balance;

balance = 100;
buffer[10] += 200;

if (balance < 0) {
    ...
}
```

```c
map_critical(&balance);
temp1 = 100;
cstore(&balance, temp1);
temp = load((buffer+40));
store((buffer+40), temp+200);
temp2 = cload(&balance);
if (temp2 > 0) {
    ...
}

balance = 100;
buffer[10] += 200;
...
```

---

Normal Mem

- 0
- 100
- 100
- 300
- 100

Critical Mem

- 0
- 100
- 100
- 100
- 100
Third-party Libraries/Untrusted Code

- Library code does not need to be critical memory aware
  - If library does not update critical data, no changes required
- If library needs to modify critical data
  - Allow normal stores to critical memory in library
  - Explicitly “promote” on return
- Copy-in, copy-out semantics

```c
int balance = 100;
...
library_foo(&balance);
promote balance;
...
```

```c
// arg is not critical int *
void library_foo(int *arg) {
  *arg = 10000;
  return;
}
```
Samurai: Heap-based Critical Memory

- Software critical memory for heap objects
  - Critical objects allocated with crit_malloc, crit_free

Approach

- Replication – base copy + 2 shadow copies
- Redundant metadata
  - Stored with base copy, copy in hash table
  - Checksum, size data for overflow detection
- Robust allocator as foundation
  - DieHard, unreplicated
  - Randomizes locations of shadow copies
Samurai Implementation

- Two replicas
- Shadow pointers in metadata
- Randomized to reduce correlated errors

**Update**

**Base Object**

**Replica 1**

**Replica 2**

**Heap**

**Vote**

Critical load checks 2 copies, detects/repairs on mismatch

- Metadata protected with checksums/backup
- Protection is only probabilistic

Regular store causes memory error!
Samurai Experimental Results

- **Implementation**
  - Automated Phoenix pass to instrument loads and stores
  - Runtime library for critical data allocation/de-allocation (C++)

- Protected critical data in 5 applications (mostly SPEC)
  - Chose data that is crucial for end-to-end correctness of program
  - Evaluation of performance overhead by instrumentation
  - Fault-injections into critical and non-critical data (for propagation)

- Protected critical data in libraries
  - **STL List Class**: Backbone of list structure (link pointers)
  - **Memory allocator**: Heap meta-data (object size + free list)
Samurai Performance Overheads

Performance Overhead

- Baseline
- Samurai

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpr</td>
<td>1.03</td>
</tr>
<tr>
<td>crafty</td>
<td>1.08</td>
</tr>
<tr>
<td>parser</td>
<td>1.01</td>
</tr>
<tr>
<td>rayshade</td>
<td>1.08</td>
</tr>
<tr>
<td>gzip</td>
<td>2.73</td>
</tr>
</tbody>
</table>

Ben Zorn, Microsoft Research

Tolerating and Correcting Memory Errors in C and C++
Samurai: STL Class + WebServer

**STL List Class**
- Modified memory allocator for class
- Modified member functions `insert`, `erase`
- Modified custom iterators for list objects
- Added a new call-back function for direct modifications to list data

**Webserver**
- Used STL list class for maintaining client connection information
- Made list critical – one thread/connection
- Evaluated across multiple threads and connections
- Max performance overhead = 9%
Samurai: Protecting Allocator Metadata

Performance Overheads

Average = 10%

Kingsley  Samurai

espresso  cfrac  p2c  Lindsay  Boxed-Sim  Mudlle  Average
Conclusion

- Programs written in C / C++ can execute safely and correctly despite memory errors

- Research vision
  - Improve existing code without source modifications
  - Reduce human generated patches required
  - Increase reliability, security by order of magnitude

- Current projects
  - **DieHard / Exterminator**: automatically detect and correct memory errors (with high probability)
  - **Critical Memory / Samurai**: enable local reasoning, allow selective hardening, compatibility
  - **Tolerace**: replication to hide data races
Hardware Trends (1) Reliability

- Hardware transient faults are increasing
  - Even type-safe programs can be subverted in presence of HW errors
    - Academic demonstrations in Java, OCaml
  - Soft error workshop (SELSE) conclusions
    - Intel, AMD now more carefully measuring
    - “Not practical to protect everything”
    - Faults need to be handled at all levels from HW up the software stack
  - Measurement is difficult
    - How to determine soft HW error vs. software error?
    - Early measurement papers appearing
Hardware Trends (2) Multicore

- DRAM prices dropping
  - 2Gb, Dual Channel PC 6400 DDR2 800 MHz $85

- Multicore CPUs
  - **Quad-core** Intel Core 2 Quad, AMD Quad-core Opteron
  - **Eight core** Intel by 2008?

- **Challenge:**
  How should we use all this hardware?
Additional Information

- **Web sites:**
  - Ben Zorn: [http://research.microsoft.com/~zorn](http://research.microsoft.com/~zorn)

- **Publications**
  - Emery D. Berger and Benjamin G. Zorn, "DieHard: Probabilistic Memory Safety for Unsafe Languages", *PLDI’06*.
Backup Slides
DieHard: Probabilistic Memory Safety

- Collaboration with Emery Berger
- Plug-compatible replacement for malloc/free in C lib
- We define “infinite heap semantics”
  - Programs execute as if each object allocated with unbounded memory
  - All frees ignored
- Approximating infinite heaps – 3 key ideas
  - Overprovisioning
  - Randomization
  - Replication
- Allows analytic reasoning about safety
Overprovisioning, Randomization

Expand size requests by a factor of $M$ (e.g., $M=2$)

```
1 2 3 4 5
```

$\Pr(\text{write corrupts}) = \frac{1}{2}$ ?

Randomize object placement

```
4 2 3 1 5
```

$\Pr(\text{write corrupts}) = \frac{1}{2}$ !
Replicating a process with different randomization seeds.

- Broadcast input to all replicas.
- Compare outputs of replicas, kill when replica disagrees.

**Diagram:**

- P1: 1 3 2 5 4
- P2: 4 3 1 5 2
- P3: 5 2 1 4 3

**Voter**
DieHard Implementation Details

- Multiply allocated memory by factor of M

Allocation

- Segregate objects by size (log2), bitmap allocator
- Within size class, place objects randomly in address space
  - Randomly re-probe if conflicts (expansion limits probing)
- Separate metadata from user data
- Fill objects with random values – for detecting uninit reads

Deallocation

- Expansion factor => frees deferred
- Extra checks for illegal free
Over-provisioned, Randomized Heap

Segregated size classes

\[ L = \text{max live size} \leq \frac{H}{2} \]

\[ F = \text{free} = H - L \]

\[ H = \text{max heap size, class } i \]

- Static strategy pre-allocates size classes
- Adaptive strategy grows each size class incrementally

Object size = 8

Object size = 16

\( 2 \ 4 \ 5 \ 3 \ 1 \ 6 \ ... \)
Randomness enables Analytic Reasoning
Example: Buffer Overflows

\[
\text{Pr(Mask Buffer Overflow)} = 1 - \left[ 1 - \left( \frac{F}{H} \right)^{Obj} \right]^k
\]

- \( k = \# \text{ of replicas, } Obj = \text{size of overflow} \)
- With no replication, \( Obj = 1 \), heap no more than 1/8 full: 
  \[
  \text{Pr(Mask buffer overflow), } = 87.5\%
  \]
- 3 replicas: \( \text{Pr(\text{ibid})} = 99.8\% \)
DieHard CPU Performance (no replication)

Runtime on Windows

Normalized runtime

<table>
<thead>
<tr>
<th></th>
<th>cfrac</th>
<th>espresso</th>
<th>lindsay</th>
<th>p2c</th>
<th>roboop</th>
<th>Geo. Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc</td>
<td>1.2</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>DieHard</td>
<td>0.9</td>
<td>1.1</td>
<td>1.2</td>
<td>1.2</td>
<td>0.8</td>
<td>1.1</td>
</tr>
</tbody>
</table>
DieHard CPU Performance (Linux)

Alloc-intensive

General-purpose

Normalized runtime

alloc-intensive

general-purpose

Geo. Mean

Geo. Mean

malloc

GC

DieHard (static)

DieHard (adaptive)
Correctness Results

- Tolerates high rate of synthetically injected errors in SPEC programs
- Detected two previously unreported benign bugs (197.parser and espresso)
- Successfully hides buffer overflow error in Squid web cache server (v 2.3s5)
- But don’t take my word for it…
Experiments / Benchmarks

- **vpr**: Does place and route on FPGAs from netlist
  - Made routing-resource graph critical
- **crafty**: Plays a game of chess with the user
  - Made cache of previously-seen board positions critical
- **gzip**: Compress/Decompresses a file
  - Made Huffman decoding table critical
- **parser**: Checks syntactic correctness of English sentences based on a dictionary
  - Made the dictionary data structures critical
- **rayshade**: Renders a scene file
  - Made the list of objects to be rendered critical
Related Work

- Conservative GC (Boehm / Demers / Weiser)
  - Time-space tradeoff (typically >3X)
  - Provably avoids certain errors
- Safe-C compilers
  - Jones & Kelley, Necula, Lam, Rinard, Adve, …
  - Often built on BDW GC
  - Up to 10X performance hit
- N-version programming
  - Replicas truly statistically independent
- Address space randomization (as in Vista)
- Failure-oblivious computing [Rinard]
  - Hope that program will continue after memory error with no untoward effects