Software Fault Tolerance for

Type-unsafe Languages C/C++

Ben Zorn
Microsoft Research

In collaboration with
Emery Berger, Univ. of Massachusetts
Karthik Pattabiraman, Univ. of Illinois, UC
Vinod Grover, Darko Kirovski, Microsoft Research
Motivation

- Consider a shipped C program with a memory error (e.g., buffer overflow)
  - By language definition, “undefined”
  - In practice, assertions turned off – mostly works
    - I.e., data remains consistent

- What if you know it has executed an illegal operation?
  - Raise an exception?
  - Continue unsoundly (failure oblivious computing)
    - Continue with well-defined semantics (Ndure)
Ndure Project Vision

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

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

- Enable trading resources for robustness
  - More memory implies higher reliability
Focus on Heap Memory Errors

- Buffer overflow

```c
char *c = malloc(100);
c[101] = 'a';
```

- Dangling reference

```c
char *p1 = malloc(100);
char *p2 = p1;
free(p1);
p2[0] = 'x';
```
Ndure Project Themes

- Make existing programs more fault tolerant
  - Define semantics of programs with errors
  - Programs complete with correct result despite errors
- Go beyond all-or-nothing guarantees
  - Type checking, verification rarely a 100% solution
    - C#, Java both call to C/C++ libraries
  - Traditional engineering allows for errors by design
- Leverage flexibility in implementation semantics
  - Different runtime implementations are semantically equivalent
Approaches to Protecting Programs

- Unsound, *may* work or abort
  - Windows, GNU libc, etc.
- Unsound, *might* continue
  - *Failure oblivious* (keep going) [Rinard]
    - Invalid read => manufacture value
    - Illegal write => ignore
- Sound, *definitely aborts* (fail-safe)
  - CCured [Necula], others
- Sound and continues
  - *DieHard, Samurai*, Rx, Boundless Memory Blocks
Exploiting Implementation Flexibility

- Runtimes are allowed to pad the allocation size request

- Consider a program with an off-by-2 buffer overflow:

  ```c
  char *c = (char*) malloc(100);
  c[101] = 'a';
  ```

- Runtimes that pad by 2 or more will tolerate this error

More efficient

No padding

Infinite padding

= padding

More fault tolerant

More efficient
Outline

- Motivation
- DieHard
  - Collaboration with Emery Berger
  - Replacement for malloc/free heap allocation
  - No source changes, recompile, or patching, required
- Critical Memory / Samurai
  - Collaboration with Karthik Pattabiraman, Vinod Grover
  - New memory semantics
  - Source changes to explicitly identify and protect critical data
- Conclusion
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(write corrupts) = \( \frac{1}{2} \)

Randomize object placement

4 2 3 1 5

Pr(write corrupts) = \( \frac{1}{2} \)!
Replication

Replicate process with different randomization seeds

P1

1 3 2 5 4

P2

4 3 1 5 2

P3

5 2 1 4 3

input

Voter

Broadcast input to all replicas

Compare outputs of replicas, kill when replica disagrees
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

\[ L = \text{max live size} \leq \frac{H}{2} \]
\[ F = \text{free} = H - L \]
\[ \text{object size} = 2^{i+3} \]
\[ H = \text{max heap size, class } i \]
Randomness allows Analytic Reasoning
Example: Buffer Overflows

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

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

### Runtime on Windows

<table>
<thead>
<tr>
<th>Software</th>
<th>Normalized Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfrac</td>
<td>0.98</td>
</tr>
<tr>
<td>espresso</td>
<td>0.98</td>
</tr>
<tr>
<td>lindsay</td>
<td>1.04</td>
</tr>
<tr>
<td>p2c</td>
<td>1.25</td>
</tr>
<tr>
<td>roboop</td>
<td>1.00</td>
</tr>
<tr>
<td>Geo. Mean</td>
<td>1.00</td>
</tr>
</tbody>
</table>

![Runtime on Windows Chart](chart.png)
DieHard CPU Performance (Linux)

Runtime on Linux

- alloc-intensive
- general-purpose

Normalized runtime

<table>
<thead>
<tr>
<th>Application</th>
<th>malloc</th>
<th>GC</th>
<th>DieHard</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfrac</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>espresso</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lindsay</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>roboop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geo. Mean</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>164 gzip</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>175 vpr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>176 gcc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>181 mcf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>186 crafty</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>197 parser</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>252 eon</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>253 permilok</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>254 gap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>255 vortex</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256 bzip2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300 twolf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geo. Mean</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Other Results

- **Correctness**
  - Tolerates high rate of synthetically injected errors in SPEC programs
  - Detected two previously unreported bugs (197.parser and espresso)
    - Uninitialized reads
  - Successfully hides buffer overflow error in Squid web cache server (v 2.3s5)
  - Tolerates crashing errors in FireFox browser

- **Performance**
  - With 16-way replication on Sun multiproc, execution takes 50% longer than single replica
Caveats

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

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

- DieHard requires non-deterministic behavior to be made deterministic (on input, gettimeofday(), etc.)

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

- DieHard exists, is available for download
  - Implemented by Emery Berger, UMass.
  - [http://www.cs.umass.edu/~emery/diehard/](http://www.cs.umass.edu/~emery/diehard/)

- You can try DieHard right now
  - Possible to replace Windows / Linux allocators
    - Requires no changes to original program
    - Non-replicated version
  - Applied to FireFox browser
    - Video on the web site
    - Hardens against heap-based exploits

- Biggest perf impact is memory usage
Outline

- Motivation
- DieHard
  - Collaboration with Emery Berger
  - Replacement for malloc/free heap allocation
  - No source changes, recompile, or patching, required
- Critical Memory / Samurai
  - Collaboration with Karthik Pattabiraman, Vinod Grover
  - New memory semantics
  - Source changes to explicitly identify and protect critical data
- Conclusion
Critical Memory Motivation

- C/C++ programs vulnerable to memory errors
  - Software errors: buffer overflows, etc.
  - Hardware transient errors: bit flips, etc.
  - Increasingly a problem due to process shrinking, power

- Critical memory goals:
  - Harden programs from both SW and HW errors
  - Allow local reasoning about memory state
  - Allow selective, incremental hardening of apps
  - Provide compatibility with existing libraries, applications
Main Idea: Data-centric Robustness

- Critical memory
  - Some data is more important than other data
  - Selectively protect that data from corruption

- Examples
  - Account data, document contents are critical
    // UI data is not
  - Game score information, player stats, critical
    // rendering data structures are not

Ben Zorn, Microsoft Research
Critical Memory Semantics

- Conceptually, critical memory is parallel and independent of normal memory
- Critical memory requires special allocate/deallocate and read/write operations
  - critical_store (cstore) – only way to consistently update critical memory
  - critical_load (cload) – only way to consistently read critical memory
- Critical load/store have priority over normal load/store
- Normal loads still see the value of critical memory
Critical Memory Benefits

- Associate critical property with types:
  - Easy to use, minimal source mods
- Allows local reasoning
  - External libraries, code cannot modify critical data
- Tolerates memory errors
  - Non-critical overflows cannot corrupt critical values
- Allows static analysis of program subset
  - Critical subset of program can be statically checked independently
- Additional checking on critical data possible

```c
int x, y, buffer[10];
critical int health = 100;

third_party_lib(&x, &y);
buffer[10] = 10000;

// health still == 100

if (health < 0) {
    die();
} else {
    x += 10;
    y += 10;
}
```
Examples

cstore health, 100
...
cloud health returns 100
load health returns 100

\[\text{store health, 10000} \]
(applications should not do this)

\[\text{load health returns 10000} \]
(depends on semantics)

\[\text{cloud health returns 100} \]
(possibly triggers exception)
Which Loads/Stores are Critical?

- All references that can read/write critical data
  - Needs to be “may-alias” for correctness
  - Must be close to the set of “must-alias” for coverage

- One approach – critical types
  - Marks an entire type as critical
  - Type-safety of subset of program that manipulates critical data
  - Rest of program can be type-unsafe
Third-party Libraries/Untrusted Code

- Library code does not need to be critical memory aware
  - If library does not modify critical data, no changes required
- If library modifies critical data
  - Allow normal stores to critical memory in library
  - Follow by a “promote”
    - Makes normal memory value critical

```c
#include <critical.h>

int health = 100;
...
lifetime_foo(&health);
promote health;
...

// arg is not critical int *
void library_foo(int *arg)
{
  *arg = 10000;
  return;
}
```
Samurai: SCM Implementation

- 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
  - Maps address to size class
  - Randomizes locations of shadow copies
Implementation

cstore health, 100

... 
cload health returns 100
load health returns 100

store health, 10000...
load health returns 10000
cload health returns 100

Ben Zorn, Microsoft Research
Software Fault Tolerance in C/C++
Samurai Experimental Results

- Prototype implementation of critical memory
  - Fault-tolerant runtime system for C/C++
  - Applied to heap objects
  - Automated Phoenix compiler pass

- Identified critical data for five SPECint applications
  - Low overheads for most applications (less than 10%)

- Conducted fault-injection experiments
  - Fault tolerance significantly improved over based code
  - Low probability of fault-propagation from non-critical data to critical data for most applications
  - No new assertions or consistency checks added
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**: Compresses/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
Results (Performance)

Performance Overhead

Baseline
Samurai

Benchmark
vpr
crafty
parser
rayshade
gzip

Slowdown

1.03
1.08
1.01
1.08
2.73
Fault Injection Methodology

- Injections into critical data
  - Corrupted objects on DieHard heap, one at a time
  - Injected more faults into more populated heap regions (Weighted fault-injection policy)
  - Outcome: success, failure, false-positive

- Injections into non-critical data
  - Measure propagation to critical data
  - Corrupted results of random store instructions
  - Compared memory traces of verified stores
  - Outcomes: control error, data error, pointer error
Fault Injection into Critical Data (vpr)

Fault Injections into vpr (with Samurai)

Fault Injections into vpr (without Samurai)
## Fault Injection into Non-Critical Data

<table>
<thead>
<tr>
<th>App</th>
<th>Number of Trials</th>
<th>Control Errors</th>
<th>Data Errors</th>
<th>Pointer Errors</th>
<th>Assertion Violations</th>
<th>Total Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpr</td>
<td>550 (199)</td>
<td>0</td>
<td>203 (0)</td>
<td>1 (0)</td>
<td>2 (2)</td>
<td>203 (0)</td>
</tr>
<tr>
<td>crafty</td>
<td>55 (18)</td>
<td>12 (7)</td>
<td>9 (3)</td>
<td>4 (3)</td>
<td>0</td>
<td>25 (13)</td>
</tr>
<tr>
<td>parser</td>
<td>500 (380)</td>
<td>0</td>
<td>3 (1)</td>
<td>0</td>
<td>0</td>
<td>3 (1)</td>
</tr>
<tr>
<td>rayshade</td>
<td>500 (68)</td>
<td>0</td>
<td>5 (1)</td>
<td>0</td>
<td>1 (1)</td>
<td>5 (1)</td>
</tr>
<tr>
<td>gzip</td>
<td>500 (239)</td>
<td>0</td>
<td>1 (1)</td>
<td>2 (2)</td>
<td>157 (157)</td>
<td>3 (3)</td>
</tr>
</tbody>
</table>
Samurai Summary

- **Critical memory**
  - Local reasoning about data consistency
  - Selective protection of application data
  - Compatible with existing libraries

- **Samurai runtime**
  - CM for heap-allocated data
  - Fault tolerance for C/C++ programs

- **Future work**
  - Uses for concurrency (integration with STM)
  - Applications to security, performance optimizations, static analysis, etc.
  - Better language integration
Conclusion

- Programs written in C can execute safely, despite memory errors with little or no source changes

- **Vision**
  - Improve existing code with little or no change
  - Reduce number of patches required
  - More memory => more reliable

- **Ndure project investigates possible approaches**
  - DieHard: overprovisioning + randomization + replicas = **probabilistic memory safety**
  - Critical Memory / Samurai: protect important data

- **Hardware trends**
  - More processors, more memory, more transient errors
Hardware Trends

- 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
Power to Spare

- DRAM prices dropping
  - 1GB < $160

- SMT & multi-core CPUs
  - **Dual-core** – Intel Pentium D & Xeons, Sun UltraSparc IV, IBM PowerPC 970MP (G5)
  - **Quad-core** Sparcs (2006), Intels and AMD Opterons (2007); more coming

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

- Publications

- Acknowledgements
  - Emery Berger, Mike Hicks, Pramod Joisha, and Shaz Quadeer
DieHard 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

- Failure-oblivious computing [Rinard]
  - Hope that program will continue after memory error with no untoward effects
Samurai Related Work

- **Address-Space Protection**
  - Virtual memory, Mondrian Memory Protection
  - Kernel extensions [SPIN, Vino], Software Fault Isolation

- **STM [Herlihy, Harris, Adl-Tabatabi]**
  - Strong atomicity for Java programs [Hindman, Grossman]

- **Memory Safety**
  - C-Cured, Cyclone, Jones-Kelley, CRED, Dhurjati-Adve
  - Singularity approach, Pittsfield

- **Error-Tolerance**
  - Rx, Failure-oblivious computing, Diehard
  - N-version programming, Recovery Blocks
  - Rio File Cache, Application-specific recovery
How to Decide What is Critical?

- Data that is important for correct execution of application or data that is required to restart the application after a crash
  - Banking application: Account data critical; GUI, networking data not critical
  - Web-server: Table of connections critical; connection state data may not be critical
  - Word-processor/Spreadsheet: Document contents critical; internal data structures not critical
  - E-Commerce application: Credit card data/shopping cart contents more critical than user-preferences
  - Game: User state such as score, level critical; state of game world not critical
Critical Memory Advantages

- Requires only accesses to critical-data to be type-safe/annotated
  - No runtime checks on non-critical accesses
- Can be deployed in an incremental fashion
  - Versus all-or-nothing approach of systems such as CCured
- Protection even in presence of unsafe/third-party library code, without requiring changes to library function or aborting upon an error
  - SFI requires modifications to library source/binary
- Amenable to possible hardware implementation
Critical Memory Limitations

- Errors in non-critical data can propagate to critical data
  - Control-flow errors (does not replace control-flow checking)
  - Data-consistency errors (assumes existence of executable assertions and consistency checks)
  - Occurred rarely in random fault-injection experiments

- Malicious attackers
  - No attempt made to hide location of shadow copies
    - Protection from adversary requires more mechanisms
  - Can exploit memory errors in non-critical data
**Samurai Operations**

- **Critical store**
  - Compute base address of object
  - Check if object is valid
  - Follow shadow pointers in metadata
  - Update replicas with stored contents

- **Critical load**
  - Compute base address of object
  - Check if object is valid
  - Follow shadow pointers in metadata
  - Check object with replicas
  - Fix any errors found by voting on a per-byte basis

---

Ben Zorn, Microsoft Research

Software Fault Tolerance in C/C++
Samurai Operations (continued)

- **Critical malloc**
  - Allocates 3 objects with diehard
  - Initializes metadata of parent object with shadow pointers
  - Set valid bits of object
  - Return base pointer to user

- **Critical free**
  - Free all 3 copies on diehard heap
  - Reset metadata of object
  - Reset valid bits of object
Heap Organization (BiBOP)

- Used in DieHard, PHKmalloc

- Allows mapping internal pointer to base object
  - Heap partitioned into pages of fixed size
  - Size classes of size $2^n$
  - Address computation to recover base pointer

$$\text{Base} = ( (\text{Ptr} - \text{Start}_8) / 8 ) \times 8$$

- Useful for checking overflow as well
Considerations and Optimizations

- **Considerations**
  - Metadata itself protected from memory errors using checksums (backup copy in protected hash table)
  - Consistency checks in implementation
    - Bounds checking critical accesses

- **Optimizations**
  - Cache frequent metadata lookups for speed
  - Compare with only one shadow on critical loads
    - Periodically switch pointers to prevent error accumulation
  - Adaptive voting strategy for repairing errors
    - Exponential back-off based on object size
    - Mainly used for errors in large objects