DieHard: Probabilistic Memory Safety for Unsafe Languages

Emery D. Berger and Benjamin G. Zorn
PLDI'06

Presented by Uri Kanonov
23.02.2014

Outline

- Introduction
- Suggested Solution
- Evaluation
- Related work
- Conclusions
- Musings…
- Discussion

Introduction

- Interested in “unsafe” languages: C/C++
- Why are those languages popular?
  - Native code is faster than interpreted code
  - Allow for more efficient optimizations
  - Fine grained control (memory/execution)
  - Can do a lot of hacky stuff !

Resulting Problems

- Programmers take control of (almost) everything (memory, resources, code flow…)
- But they often…
  - Forget to handle the resources properly
  - Are unaware of their runtime environment (memory layout, how to the heap works)
  - Write poor code that leads to bugs ☺
- End result
  - Security vulnerabilities
  - Crashes

Goal

- Efficiently detect /prevent such bugs
- Multiple approaches:
  - Detect statically
  - Countermeasures to avoid the bugs ←
  - Detect at runtime and
    - Tolerate ←
    - Perform a controlled crash ←
  - Ignore ☺

DieHard

Proposed Solution: DieHard

- Takes on a "hardening" approach:
  - Dangling pointers **Avoiding + Tolerating**
  - Buffer overflows **Avoiding + Tolerating**
  - Heap metadata overwrites **Avoiding**
  - Uninitialized reads **Detecting and crashing**
  - Invalid frees **Tolerating**
  - Double frees **Tolerating**

DieHard

- Heap allocator based on “probabilistic memory safety”
- Ideal: an infinite heap
  - Never freeing
  - Infinite spacing
- Practical: heap *M* times larger than required

In Practice

- How allocations work?
  - Heap initialized to random data
  - Objects allocated at random locations across the heap
- Separate heap metadata
- Run multiple copies to detect uninitialized reads

Initialization

- Heap size: *M* times the needed size
- 12 regions
  - Powers of two from 8 bytes to 16KB
  - Larger objects allocated separately
  - Filled up to 1/M of its size
- Heap meta-data
  - Separate
  - Bitmap per region consisting of bit per object

Motivation

- Why use regions per object size?
  - To prevent external fragmentation
  - Knowing the region tells you the object size
  - Powers of two -> efficient calculations
- Why separate heap metadata?
  - Security

Pseudo-code

```
1   void DieHardInitHeap (int MaxHeapSize) {
2       // Initialize the random number generator
3       // with a truly random number.
4       rng.setSeed (realRandomSource);
5       // Clear counters and allocation bitmaps
6       // for each size class.
7       for (c = 0; c < NumClasses; c++) {
8           inUse[c] = 0;
9           isAllocated[c].clear();
10      }
11      // Get the heap memory.
12      heap = mmap (NULL, MaxHeapSize);
13      // REPLICATED: fill with random values
14      for (i = 0; i < MaxHeapSize; i += 4)
15              ((long *) heap)[i] = rng.next();
16      }
```

Allocation

- Allocating large objects with mmap
  - Use “guard” (no-rw) pages
- Locating empty slot in object’s region
  - Fails if region is full (OOM)
  - Expected time to find an empty slot: $\frac{1}{1-(1/M)}$
- Slot filled with random values
- Occupying entire slot even if object is smaller

Pseudo-Code

```
1  void * DieHardMalloc (size_t sz) {
2       if (sz > MaxObjectSize)
3           return allocateLargeObject(sz);
4       c = sizeClass (sz);
5       if (inUse[c] == PartitionSize / (M * sz))
6           // At threshold: no more memory.
7           return NULL;
8       do { // Probe for a free slot.
9           index = rng.next() % bitmap size;
10          if (!isAllocated[c][index]) {
11             // Found one, pick pointer corresponding to slot.
12             ptr = PartitionStart + index * sz;
13             inUse[c]++; // Mark it allocated.
14             isAllocated[c][index] = true;
15             // REPLICATED: fill with random values.
16             for (i = 0; i < getSize(c); i += 4)
17                 ((long *) ptr)[i] = rng.next();
18             return ptr;
19          }
20      } while (true);
21  }
```

Deallocation

- If address lies inside heap:
  - If "large object" object, it is deallocated
  - Otherwise, ignored
- Assertions:
  - Object offset from region start is multiple of size
  - The object must be allocated
- Eventually slot is marked as free

Pseudo-code

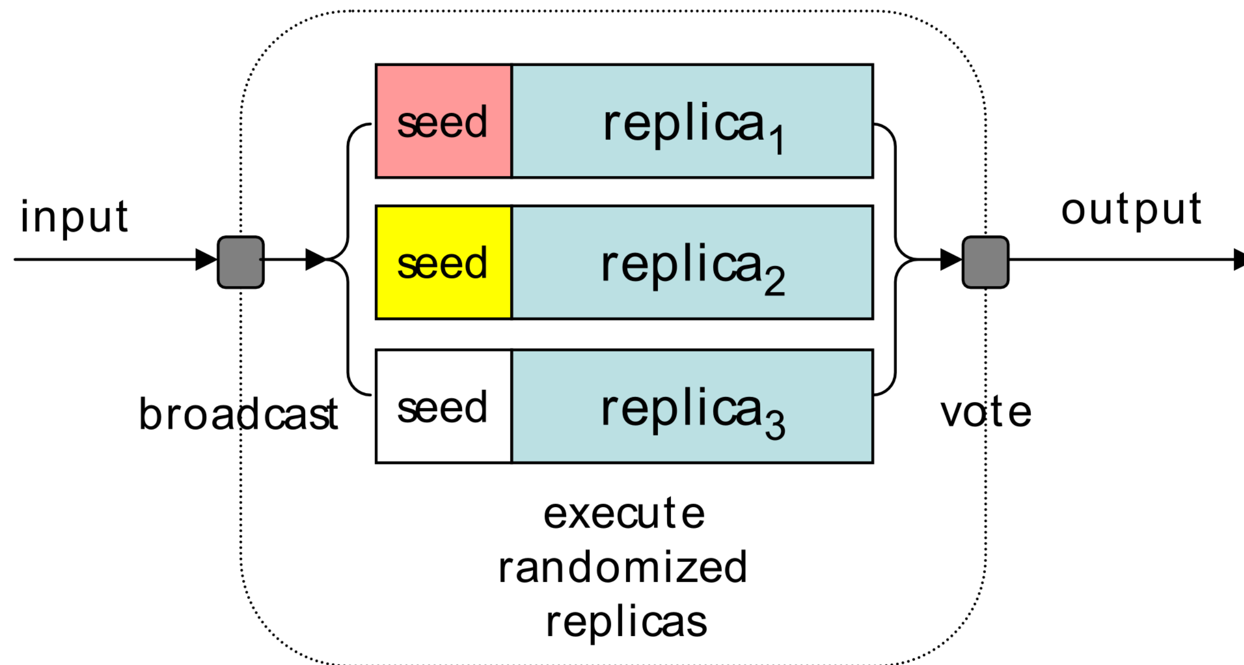
```
1  void DieHardFree (void * ptr) {
2        if (ptr is not in the heap area)
3           freeLargeObject(ptr);
4        c = partition ptr is in;
5        index = slot corresponding to ptr;
6        // Free only if currently allocated;
7        if (offset correct && isAllocated[c][index]) {
8           inUse[c]--; // Mark it free.
9           isAllocated[c][index] = false;
10       } // else, ignore
11 }
```

Secure strcpy

- Override strcpy/strncpy to prevent buffer overflows
- Doesn't mitigate other risks:
  - memcpy / memmove
  - User defined functions

```
1 void foo(char* user_input) {
2         char* buffer = (char*)malloc(100);
3         strcpy(buffer, user_input);
4 }
```

Replication



- Assumption
  - Program's output depends on data it reads
  - Uninitialized data -> different outputs amongst replicas
- Output is buffered and voted on (majority voting)

Replication (cont.)

- Non-agreeing replicas are terminated
- Implementation limitations :
  - What if a replica enters an infinite loop
  - Non-deterministic or environment dependent programs are not supported
  - Significant memory/CPU overhead

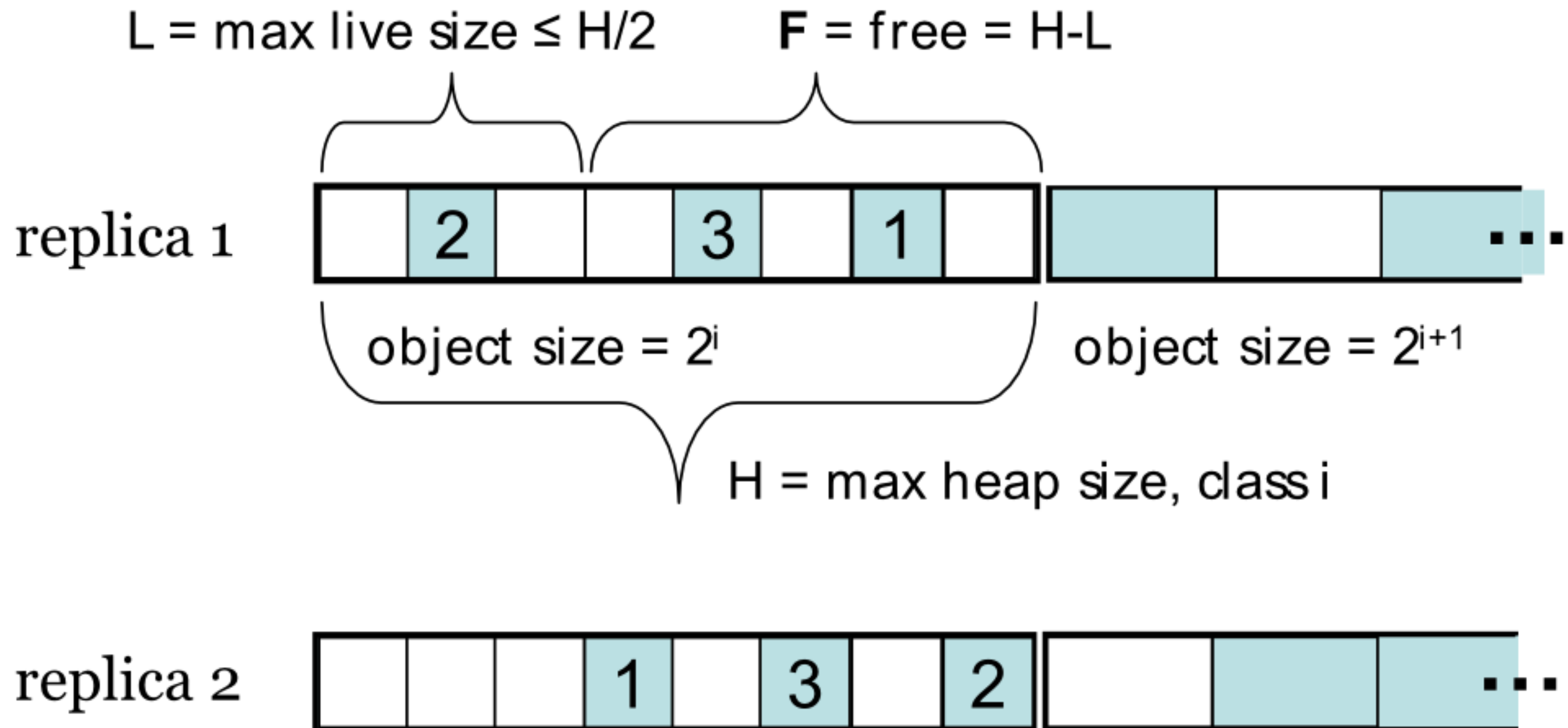
Correctness

- Does DieHard follow through on its promises?
  - Heap metadata overwrites
    - Separate metadata
  - Invalid/double frees
    - Deallocation performs the required validations
  - Uninitialized reads
    - Probabilistically
  - Dangling pointers and Buffer overflows
    - Probabilistically
- Yep…

Masking Buffer Overflows

- Lets analyze how DieHard deals with buffer overflows
- Some notations first:
  - H - Heap expansion factor
  - k - Number of replicas
  - M - Max heap size
  - L - Maximum live size $L < H/M$
  - F - Remaining free space $F = H - L$
  - O - Number of objects' worth of bytes overflowed

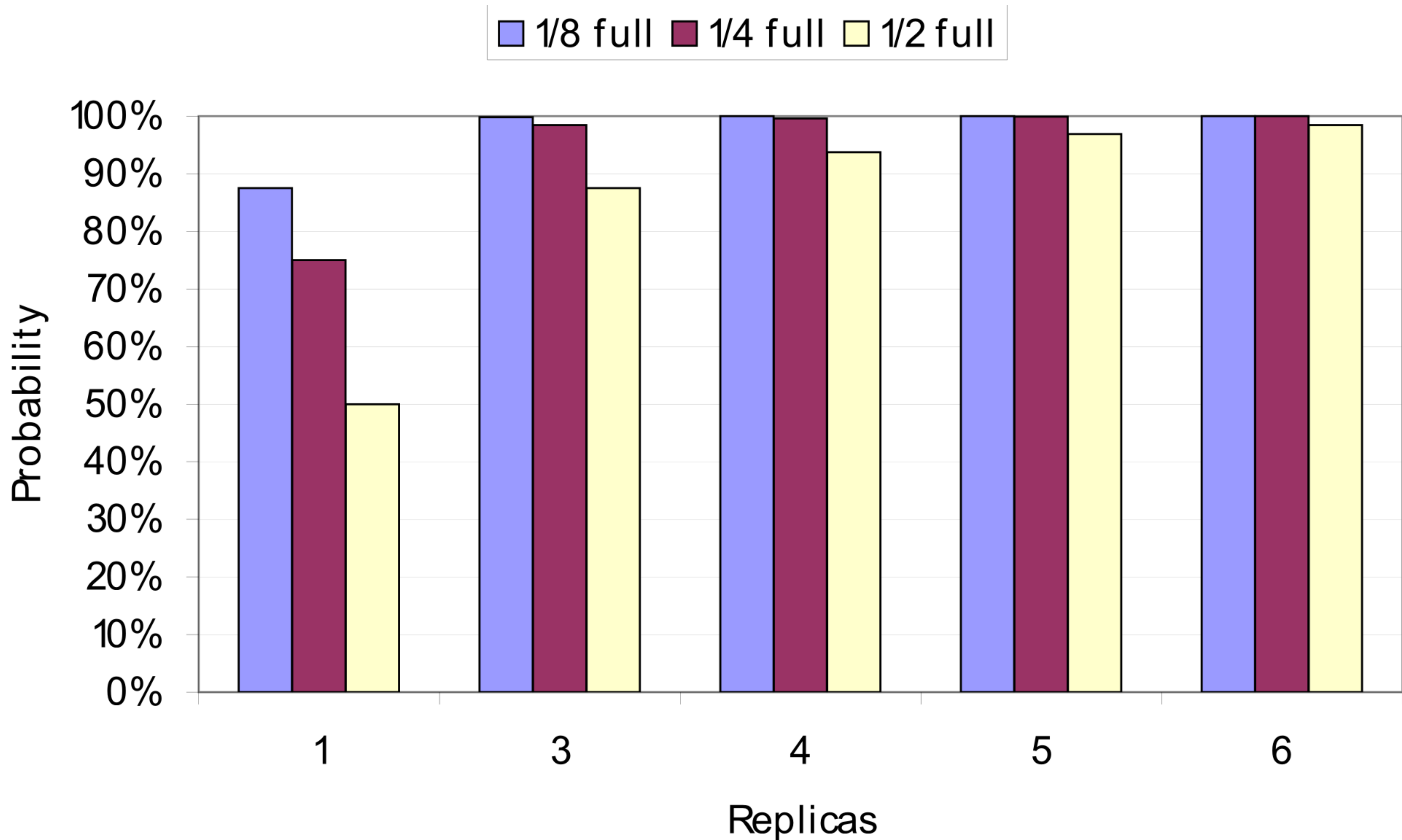
Heap Layout



Masking Buffer Overflows (cont.)

- **Theorem:** $P(NoOverflows) = 1 - \left[1 - \left(\frac{F}{H}\right)^{O}\right]^{k}$
- **Proof:**
  - Odds of O objects overwriting at least one live object are 1 minus the odds of them overwriting no live objects: $1 - \left(\frac{F}{H}\right)^{O}$
  - Masking requires that at least one replica of the k replicas not overwrite any live objects, alternatively all of them overwriting at least one live object: $1 - \left[1 - \left(\frac{F}{H}\right)^{O}\right]^{k}$

Probability of Avoiding Buffer Overflows



Runtime Complexity

- **Initialization / Deallocation**
  - No significant runtime overhead
- **Allocation:**
  - "Mild" impact due to the empty slot search
- **Accessing allocated memory**
  - No "spatial locality" -> many TLB misses
  - Need the heap to fit into the physical RAM

Memory Complexity

- Heap size
  - 12M times more memory is required
- Object size rounding
  - Up to X2 memory is used
  - Same approach used in many allocators
- Heap metadata takes up little very little space
- Segregated regions
  - Eliminate external fragmentation

Evaluation

- DieHard was evaluated on two criteria:
  - Runtime overhead (complexity)
  - Error avoidance (correctness)
- We will elaborate on each in detail

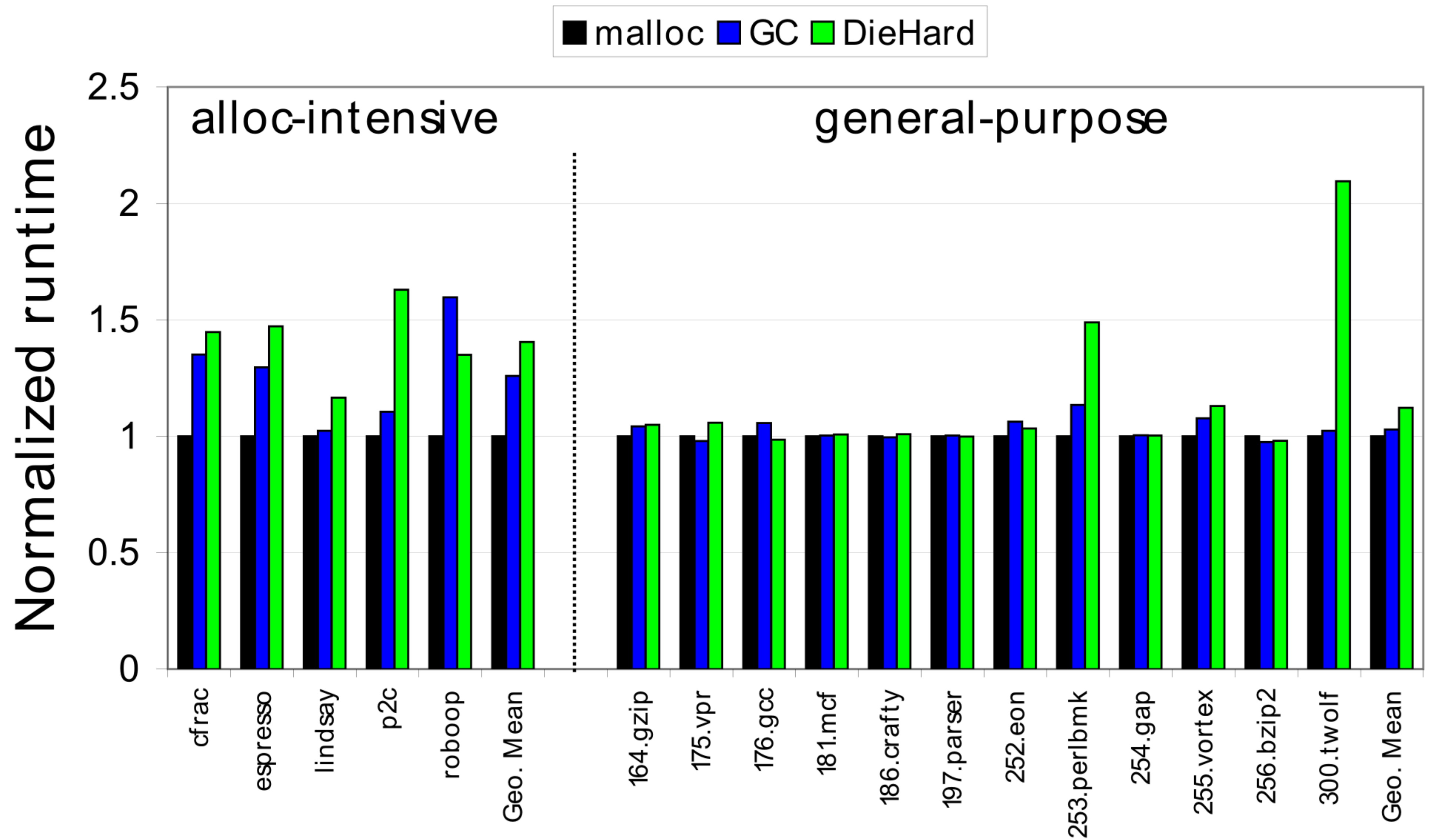
Runtime Overhead Evaluation

- Benchmark suite:
  - SPECint2000
  - Allocation-intensive benchmarks (100K - 1.7M allocations per sec)
- Heap size: 384MB with ½ available for allocation
- Operating Systems
  - StandAlone: Windows XP & Linux
  - Repliacted: Solaris

Experiments

- Linux:
  - DieHard
  - Native (GNU libc) allocator
  - Boehm-Demers-Weiser garbage collector
- Windows XP:
  - DieHard
  - Native allocator
- Solaris
  - Replicated version

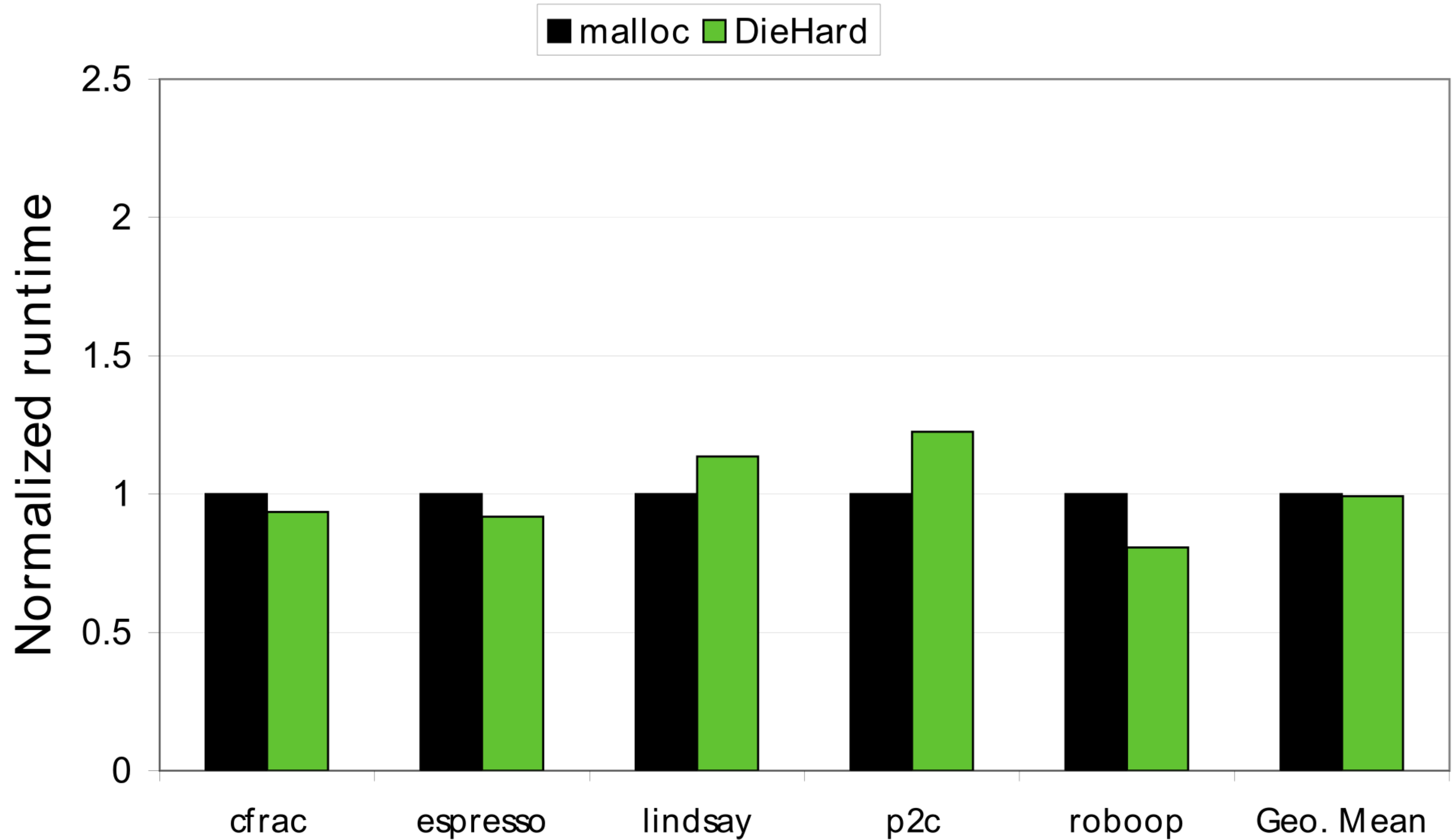
Runtime on Linux



Linux Results

- High overhead (16.5% to 63%):
  - Allocation intensive applications
  - Wide usage of different object sizes -> TLB misses
- Low overhead:
  - General purpose (SPECint2000) benchmarks

Runtime on Windows XP



Windows XP Results

- Surprise!
  - DieHard performs on average like the default allocator
- The authors' explanation
  - Windows XP's allocator is much slower than GNU libc's
  - The compiler on Windows XP (Visual Studio) produces more efficient code than g++ on Linux
- Interesting question
  - How would DieHard perform on modern Windows?

Solaris Results

- Experiment
  - Use a 16-core Solaris server
  - Run 16 replicates of the allocation-intensive benchmarks
- Results
  - One benchmark terminated by DieHard due to an uninitialized read
  - Rest of the benchmarks incurred 50% runtime overhead
  - Process creation overhead would be amortized by longer-running benchmarks

Error Avoidance – Real Scenario

- Version of the Squid web cache server containing a buffer overflow bug
- Results
  - DieHard contains this overflow
  - GNU libc allocator and the BDW collector crash
- Impressive!
  - Interesting to see DieHard pitted against more bugs

Error Avoidance – Injected Faults

- Performed on a UNIX machine
- Single allocation-intensive benchmark
- strcpy and strncpy were not overriden
- MITM'ing allocations
  - Buffer overflows: Caused by under-allocating buffers
  - Dangling pointers: Freeing an object sooner than its actual free

Dangling Pointers - Results

- One out of every two objects is freed ten allocations too early
- Results
  - Default allocator (GNU libc)
    - The benchmark failed to complete all 10 times
  - DieHard
    - Ran correctly 9 out of 10 times

Buffer Overflows - Results

- Under-allocating by 4 bytes one out of every 100 allocations for >= 32 bytes

- Results
  - Default allocator
    - 9 crashes and one infinite loop
  - DieHard
    - 10 successful runs

Evaluation Conclusions

- Runtime overhead
  - Is suitable for general purpose applications
  - Is NOT suitable for allocation-intensive ones
  - The replicated version scales well to computers with a large number of processors
- Error avoidance
  - Seems to contain well both artificial and real faults

Related work – Fail-Stop Approach

- Prototype
  - CCured / Cyclone
- Idea
  - Provide type/memory safety to C/C++ using runtime checks and static analysis
- Pros
  - May detect other errors that DieHard can't
- Cons
  - Requires code modification
  - May abort errors that DieHard can "contain"

Related work – Failure Masking

- Idea
  - Ignore illegal writes and manufacture values for uninitialized reads.
- Pros
  - May incur less overhead than DieHard
- Cons
  - May result in unpredictable program behavior

Related work - Rollback

- Prototype
  - Rx
- Idea
  - Utilize logging and rollbacks to restart programs after detectable errors (like a crash)
- Pros
  - May incur less overhead than DieHard
- Cons
  - Rollbacks aren't suitable for every program
  - Not all errors are detectable externally

Conclusions

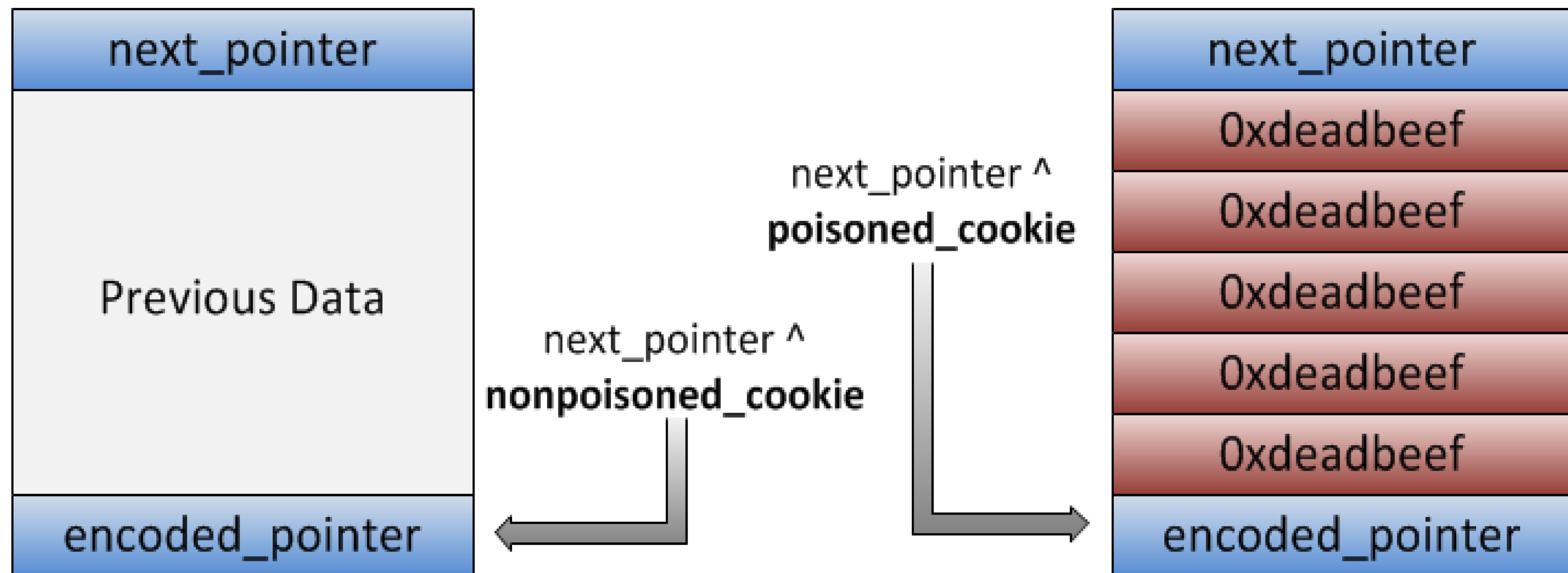
- "Probabilistic memory safety" has its merits!
  - Especially revolutionary for 2006…
- DieHard can contain avoid/contain certain errors but at a high cost
  - Not suitable for all applications
- DieHard uses many common-practice techniques
  - Separation of heap meta-data
  - Separate regions by object size

  Meaning… they work!

Musings…

- Nowadays when RAM is usually not an issue DieHard can be a suitable solution for general purpose applications
- Randomness is useful against “bugs” but not against those who try to exploit them
- Modern OS use more efficient/simple ways to protect against overflows
  - Heap cookies!
  - For example: iOS 6

iOS 6 Heap Cookies



iOS 6 Heap Cookies

- alloc( ) ensures next_pointer matches encoded pointer at end of block
  - Tries both cookies
  - If poisoned cookie matches, check whole block for modification of sentinel (0xdeadbeef) values
- Next pointer and cookie replaced by 0xdeadbeef when allocated

Questions?

Discussion

- What do you think about DieHard?
  - Is it practical?
  - Would you use it in your application?
- Is heap cookie solution secure enough?
- Any other suggestions?