Memory Safety Through Runtime Analysis

Justin Samuel for CSE 504, Spring ‘10

Instructor: Ben Livshits

(A Few) Runtime Safety Approaches

Approach	Year	Summary
SFI	1994	Software Fault Isolation.
Bounds checking C	1995	Jones and Kelly, CRED.
StackGuard	1998	Canaries.
ASLR	2001	Address Space Layout Randomization (e.g. PAX).
Program Shepherding	2002	Run through an interpreter and verify branch instructions.
PointGuard	2003	Pointers encrypted in memory and decrypted at time of use.
CFI	2005	Control-Flow Integrity.
DieHard	2005	Multiple, large heaps with different allocation.
DFI	2006	Data-Flow Integrity.
WIT	2008	Write Integrity Testing.
NaCL	2009	Native Client. Uses SFI. Performs CFI.

CFI: Control-Flow Integrity

- Ensure execution follows the control-flow graph (CFG).
  - Statically analyze binary to identify valid destinations of all control transfers.
  - Instrument code with:
    - Unique IDs at destinations.
    - Checking destination IDs before all instructions that transfer control.
- Not concerned with read or write destinations.

DFI: Data-Flow Integrity

- Restrict reads based on instructions that wrote the data.
  - Statically analyze source to identify the instructions that are allowed to write values that are read.
  - Instrument code to:
    - Maintain a table of the last instructions to write memory locations.
    - Check the last-write table on reads against computed allowed write instructions.
- Not directly concerned with control flow.

Preventing memory error exploits with WIT

Periklis Akritidis, Cristian Cada, Costin Raiciu, Manuel Costa, Miguel Castro

Microsoft Research, Cambridge UK

Example Vulnerable Code

```
1:  char cgiCommand[1024];
2:  char cgiDir[1024];
3:
4:  void ProcessCGIRequest(char* msg, int sz) {
5:      int i=0;
6:      while (i < sz) {
7:          cgiCommand[i] = msg[i];
8:          i++;
9:      }
10:
11:     ExecuteRequest(cgiDir, cgiCommand);
12: }
```

Data-modifying Commands

```
1:  char cgiCommand[1024];
2:  char cgiDir[1024];
3:
4:  void ProcessCGIRequest(char* msg, int sz) {
5:      int i=0;
6:      while (i < sz) {
7:          cgiCommand[i] = msg[i];
8:          i++;
9:      }
10:
11:     ExecuteRequest(cgiDir, cgiCommand);
12: }
```

Line 7's Modified Objects

```
1:   char cgiCommand[1024];
2:   char cgiDir[1024];
3:
4:   void ProcessCGIRequest(char* msg, int sz) {
5:       int i=0;
6:       while (i < sz) {
7:           cgiCommand[i] = msg[i];
8:           i++;
9:       }
10:
11:      ExecuteRequest(cgiDir, cgiCommand);
12: }
```

WIT: Write Integrity Testing

- Approach:
  1. Determine memory locations that individual instructions should legitimately be able to write to.
  2. Only allow a given instruction to write to those locations.
- Memory errors to protect against:
  - Buffer overflows and underflows
  - Dangling pointers
  - Double frees

Static Analysis

- Two stages:
  1. Points-to analysis
  2. Write safety analysis
- Goal: Create a color table.
  - Give each unsafe object a different color.
  - Give each write instruction the same color as the objects it can write.

Static Step 1: Points-to Analysis

- Compute the set of objects that can be modified by each program instruction.
- In the example:
  - Set **{i}** for the instructions at lines 5 and 8
  - Set **{cgiCommand}** for the instruction at line 7

```
1:  char cgiCommand[1024];
2:  char cgiDir[1024];
3:
4:  void ProcessCGIRequest(char* msg, int sz) {
5:      int i=0;
6:      while (i < sz) {
7:          cgiCommand[i] = msg[i];
8:          i++;
9:      }
10:
11:     ExecuteRequest(cgiDir, cgiCommand);
12: }
```

Static Step 2: Write Safety Analysis

- Purpose: runtime efficiency.
- For all instructions and objects, determine whether safe or unsafe.
  - **Safe instruction**: cannot violate write integrity.
    - No destination operand
    - Operand is temporary, local, or global variable.
  - **Safe object**: all instructions that can modify it are safe.

Safe

```
1:  char cgiCommand[1024];
2:  char cgiDir[1024];
3:
4:  void ProcessCGIRequest(char* msg, int sz) {
5:      int i=0;
6:      while (i < sz) {
7:          cgiCommand[i] = msg[i];
8:          i++;
9:      }
10:
11:     ExecuteRequest(cgiDir, cgiCommand);
12: }
```

Not Safe

```
1:  char cgiCommand[1024];
2:  char cgiDir[1024];
3:
4:  void ProcessCGIRequest(char* msg, int sz) {
5:      int i=0;
6:      while (i < sz) {
7:          cgiCommand[i] = msg[i];
8:          i++;
9:      }
10:
11:     ExecuteRequest(cgiDir, cgiCommand);
12: }
```

Example Color Table

- Record the color of each memory location.
- Color 0 is used for safe objects.

Color	Instructions	Objects
0	Lines 5, 8	msg, sz, i
3	Line 7	cgiCommand
4		cgiDir

Function Colors

- Compute possible indirect function calls.
- Colors assigned to functions are disjoint from those assigned to objects.
- Prevents:
  - Unsafe instructions from overwriting code.
  - Control transfers outside code regions.

Instrumentation

- Insert guards.
- Maintain color table.
- Check writes.
- Check indirect calls.

- Use information from static analysis.
- Add instrumentation during compilation.

Instrumentation

- **Insert guards.**
- Maintain color table.
- Check writes.
- Check indirect calls.

- Points-to analysis is imprecise.
  - False negatives possible
- Insert guards between unsafe objects.
- Guard objects have color 0 (safe objects).

Instrumentation

- **Insert guards.**
- Maintain color table.
- Check writes.
- Check indirect calls.

Instrumentation

- **Insert guards.**
- Maintain color table.
- Check writes.
- Check indirect calls.

- Different for the stack, heap, and global data.
- Heap allocator's header used as a guard by setting its color to 1.

Instrumentation

- **Insert guards.**
- Maintain color table.
- Check writes.
- Check indirect calls.

- Tricky guard case:
  - Function arguments written by unsafe instruction.
  - Solution: Copy argument to local variable, guard that, and rewrite instructions to refer to the copy.

Instrumentation

- Insert guards.
- **Maintain color table.**
- Check writes.
- Check indirect calls.

- 8-bit color for each 8-byte memory slot
  - Space overhead is 12.5%
  - Pad generated code.
- Instrument function prologues and epilogues to set and reset color table entries.
- Wrappers for allocation functions (`malloc`, `calloc`, `free`).

Instrumentation

- Insert guards.
- Maintain color table.
- **Check writes.**
- Check indirect calls.

- Only check writes by unsafe instructions.
- Compare color of instruction to destination operand.
  - If they do not match, raise an exception.

Instrumentation

- Insert guards.
- Maintain color table.
- Check writes.
- **Check indirect calls.**

- Lookup the color of the target function.
- Compare with the color of the indirect call instruction.
  - If they do not match, raise an exception.
- Zero last three bits of function pointer value to ensure 16-byte aligned, which should normally be the case.

Prevented Attacks

- WIT can prevent all attacks that violate write integrity.
  - Depends on precision of points-to analysis.
- If two objects have the same color, WIT may fail to detect an attack.
- Sequential overflow always prevented even if colors match.
- What about reads?

Number of Writable Objects

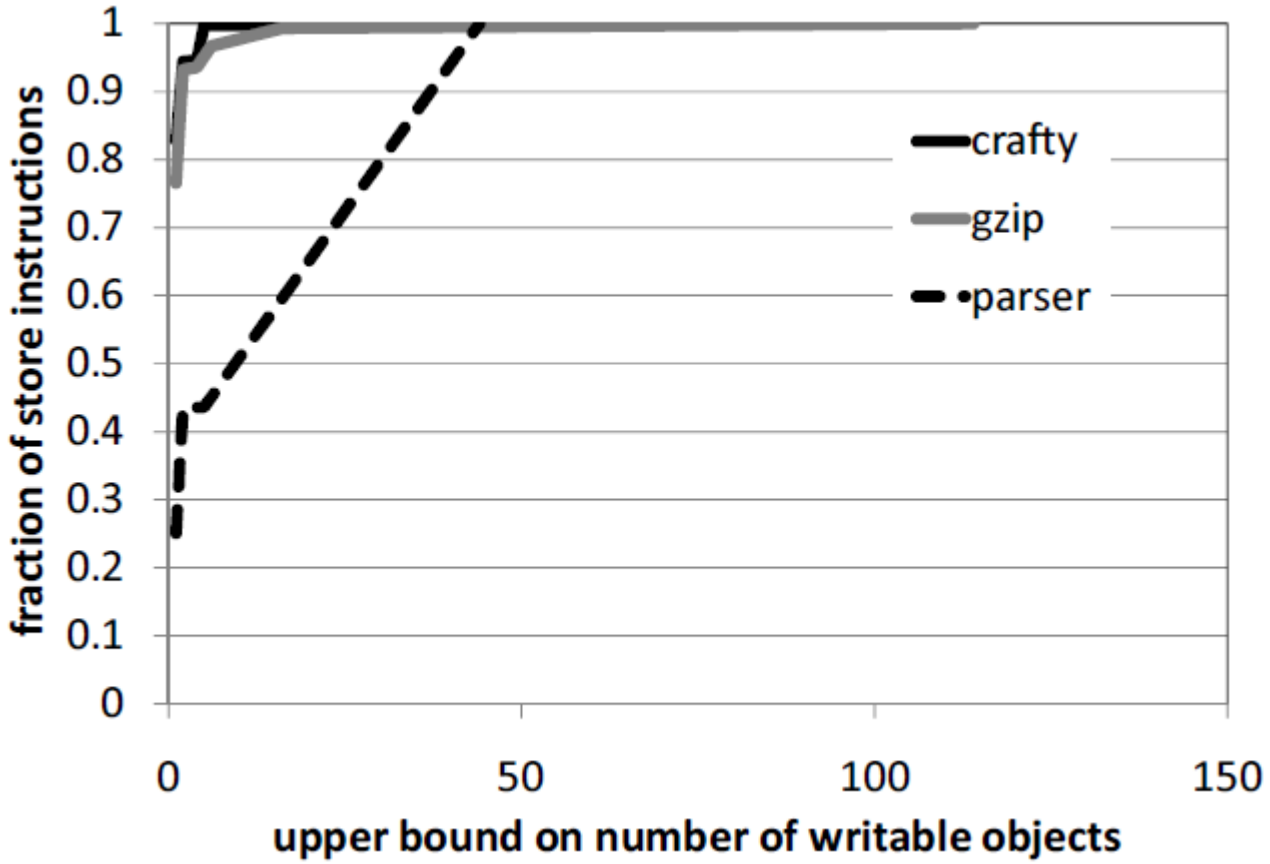


Figure 5. Cumulative distribution of the fraction of store instructions versus the upper bound on the number of objects writable by each instruction.

CPU Overhead

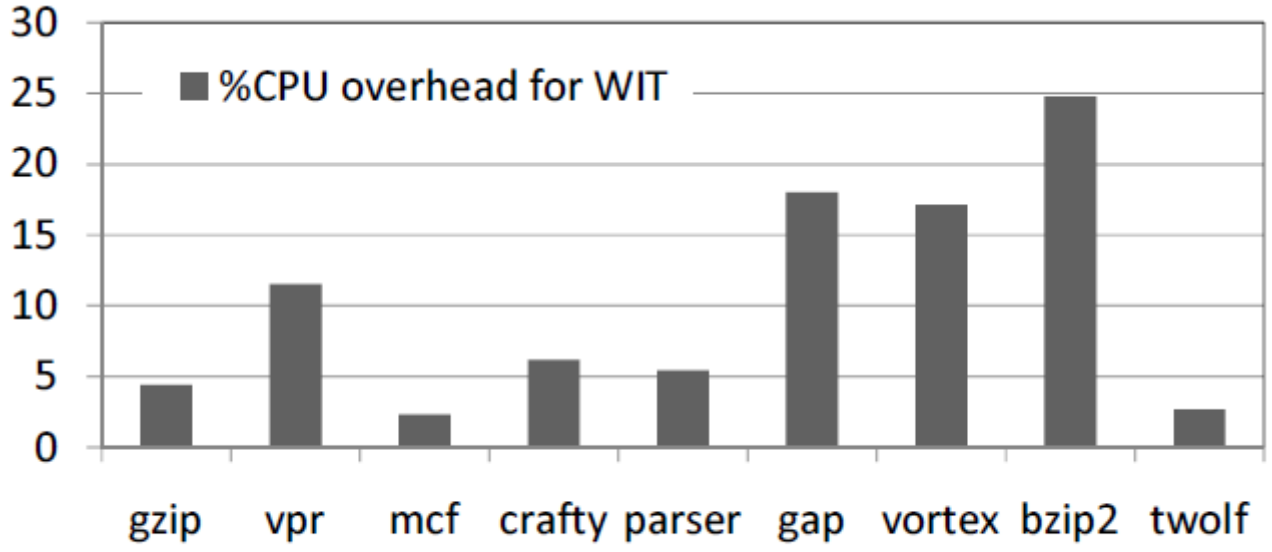


Figure 6. CPU overhead on SPEC benchmarks.

Memory Overhead

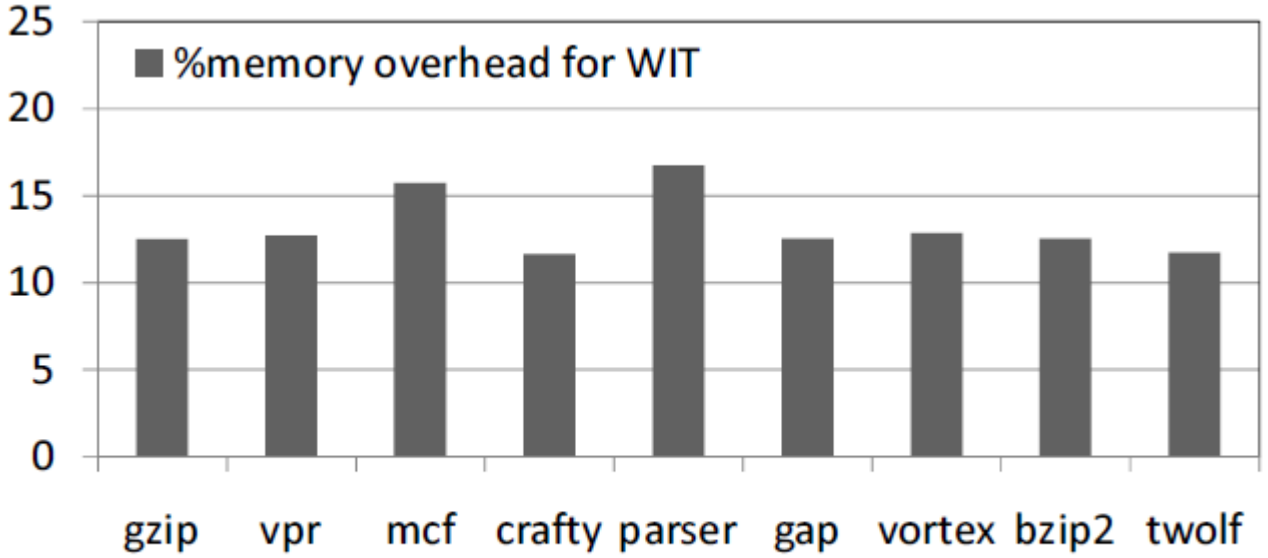


Figure 8. Memory overhead on SPEC benchmarks.

Testing with Vulnerabilities

- Successful against benchmark of 18 control-data attacks that exploit buffer overflows.
  - All but one are detected when guard object overwritten. The other is detected when a corrupted pointer is used to overwrite a return address (color 0).
- Tested with known vulnerabilities in real apps.
  - All detected when buffer overflow hit a guard object at the end of the buffer.
    - SQL Server sprintf overflow of stack buffer (Slammer).
    - Ghttpd vsprintf overflow of stack buffer.
    - Nullhttpd heap buffer overflow causes heap management data structures to be overwritten.
    - Stunnel vsprintf format string overflow of stack buffer.
    - Libpng stack buffer overflow.

Limitations

- Libraries
  - WIT as just described doesn't work for libraries.
  - WIT for libraries assigns the same well-known color to all unsafe objects allocated by libraries.
    - Will WIT work without recompiled libraries?

DieHard: Probabilistic Memory Safety for Unsafe Languages

Emery D. Berger and
Benjamin G. Zorn

Non-fatal Memory Errors

- Existing approaches either:
  - Abort when memory errors detected.
  - Continue anyways (!)
- How about detecting the memory error and allowing the program to continue correctly?

DieHard in a Nutshell

- Randomize object locations in a large heap.
- Can operate in a replicated mode.
  - Multiple replicas of the same application are run simultaneously. Require agreement on output.

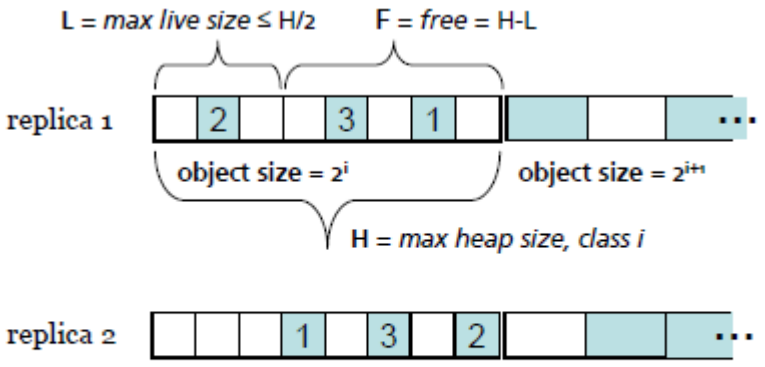


Figure 1. DieHard's heap layout. The heap is divided into separate per-size class regions, within which objects are laid out randomly. Notice the different layouts across replicas.

Randomized Object Locations

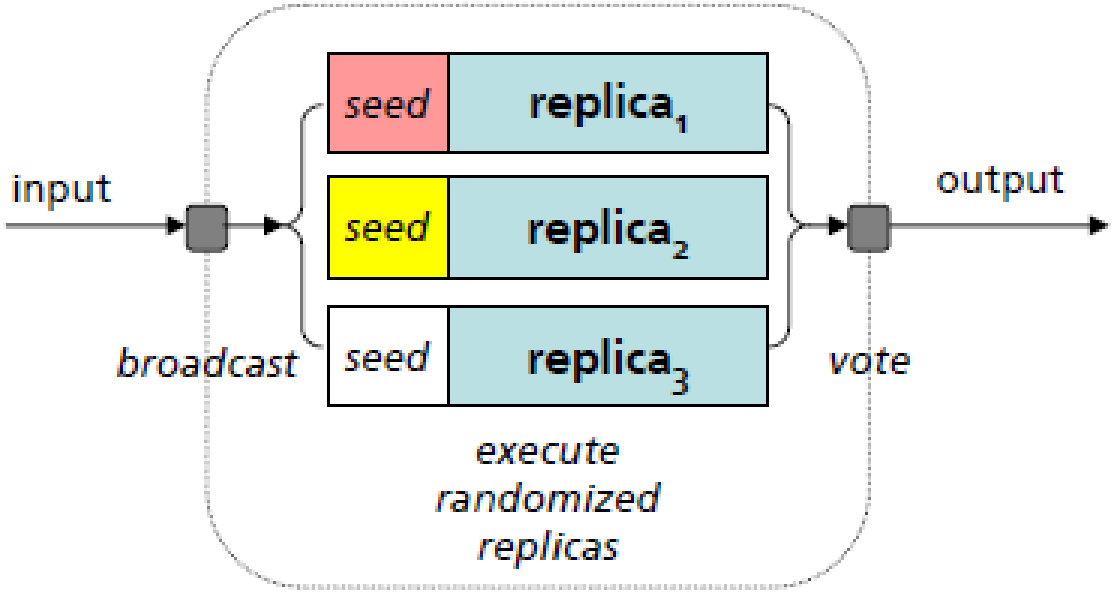
- Likely that buffer overflows end up overwriting only empty space.
- Unlikely that a newly-freed object will soon be overwritten by a subsequent allocation.

Replication

- Stand-alone DieHard cannot detect uninitialized reads.
- Solution: execute several replicas simultaneously.

Detecting Uninitialized Reads

1. Fill allocated object with random values.
2. Execute same program in multiple replicas.
3. Compare outputs.

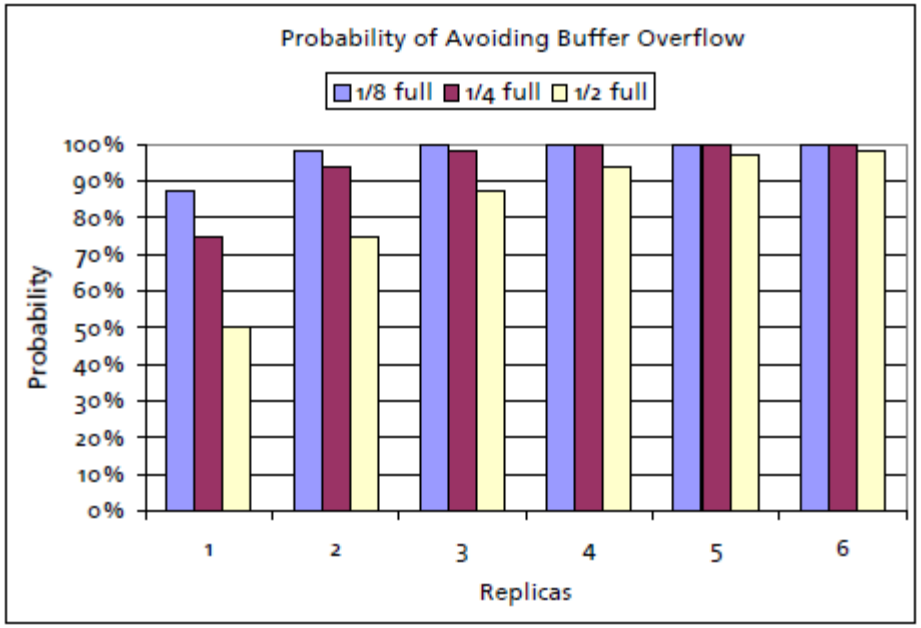


Replica Communication

- DieHard uses pipes and shared memory to communicate with replicas.
- Each replica receives stdin from and writes stdout to DieHard.
- DieHard compares output from replicas.
- Support not implemented for programs that modify filesystems or perform network I/O.
- How would you support non-deterministic programs?

Avoiding Buffer Overflows

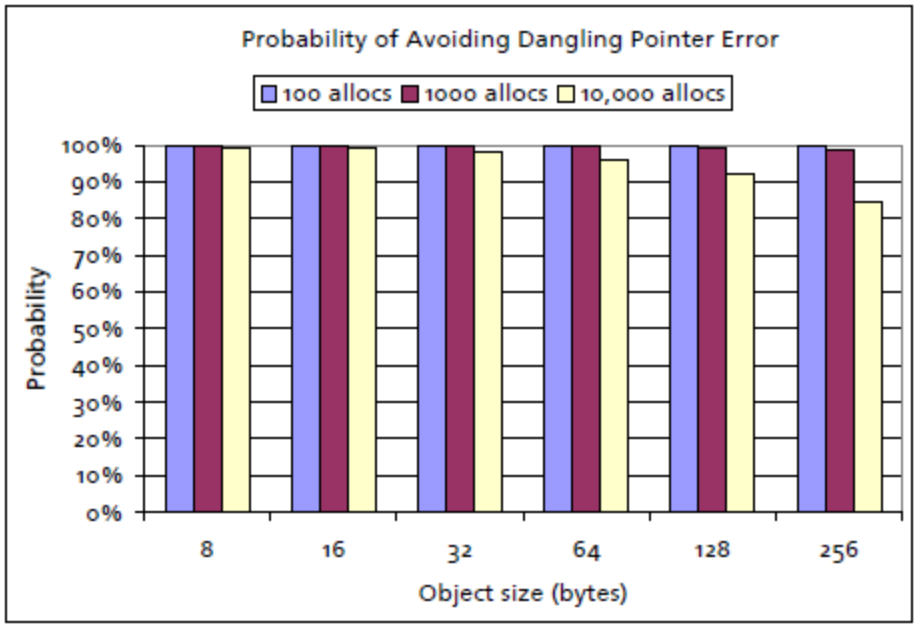
"…for our analysis, we model a buffer overflow as a write to any location in the heap."



(a) Probability of masking single-object buffer overflows for varying replicas and degrees of heap fullness.

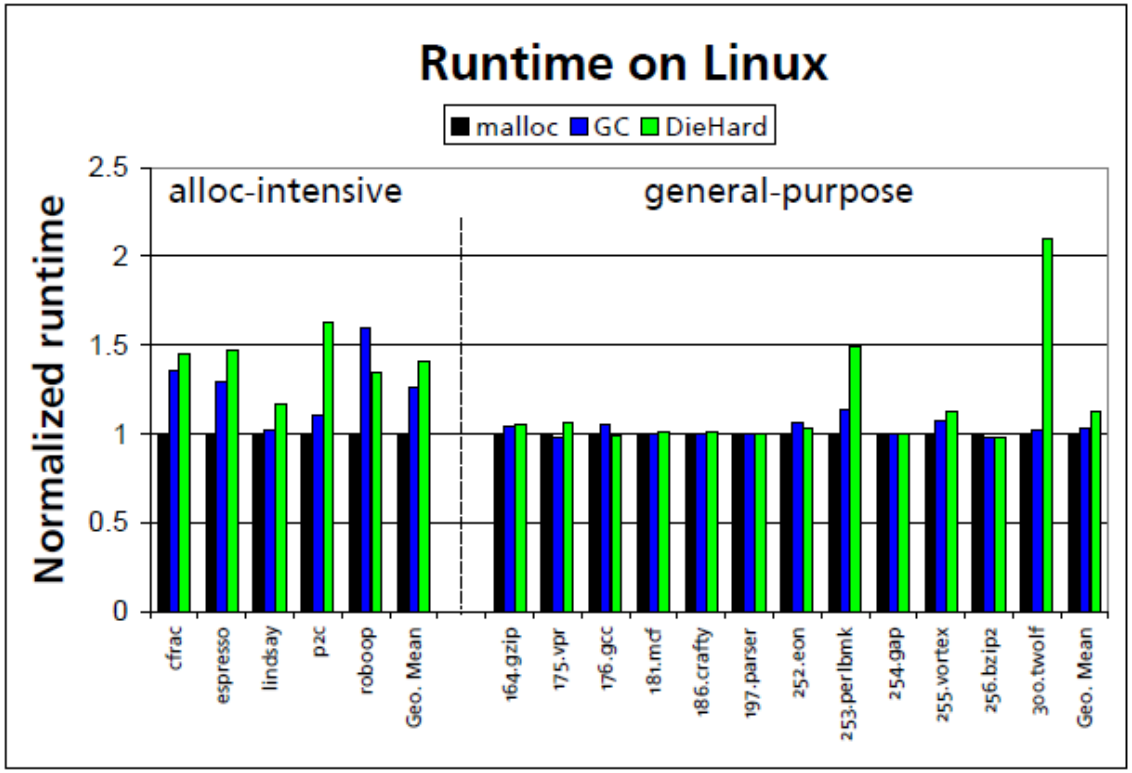
Avoiding Dangling Pointers

"…the likelihood that the object's contents are not overwritten after A intervening allocations"



(b) Probability of masking dangling pointer errors *using the stand-alone version of DieHard* in its default configuration, for varying object sizes and intervening allocations.

Runtime on Linux



(a) Linux: Performance of the default `malloc`, the Boehm-Demers-Weiser garbage collector, and DieHard (stand-alone version), across a range of allocation-intensive and general-purpose benchmark applications.

Other Approaches

Error	GNU libc [25]	BDW GC [8]	CCured [27]	Rx [30]	Failure-oblivious [31]	DieHard
heap metadata overwrites	undefined	undefined	abort	✓	undefined	✓
invalid frees	undefined	✓	✓	undefined	undefined	✓
double frees	undefined	✓	✓	✓	undefined	✓
dangling pointers	undefined	✓	✓	undefined	undefined	✓*
buffer overflows	undefined	undefined	abort	undefined	undefined	✓*
uninitialized reads	undefined	undefined	abort	undefined	undefined	abort*

Table 1. This table compares how various systems handle memory safety errors: ✓ denotes correct execution, `undefined` denotes an undefined result, and `abort` means the program terminates abnormally. See Section 8 for a detailed explanation of each system. The DieHard results for the last three errors (marked with asterisks) are probabilistic; see Section 6 for exact formulae. Note that `abort` is the only sound response to uninitialized reads.

Rx: rollback to a checkpoint, re-execute in a modified environment.

Comparison

Approach	Protections	Limitations	CPU Overhead
Memory-safe C (CCured, Cyclone)	All memory errors.	Source and runtime changes (e.g. garbage collector).	High
Taint analysis	Many memory errors.	Accuracy vs. automation.	2x WIT
Bounds checking C	All buffer overflows.	Doesn't work with all programs. Protection granularity limited by compile-time information.	Up to 12x WIT
StackGuard, et al.	Specific targets (e.g. return address).	Only defend against specific attacks.	Low
CFI	Control-flow.	Data not protected.	15-45%
DFI	Some out-of-bounds reads and writes	No guards against imprecise analysis.	104% over WIT (so, 20-50%)
WIT	Incorrect writes.	Library incompatibility.	10-25%
DieHard	Probabilistic memory safety.	Large heap, only supports simple programs.	12-109%.
Dhurjati's improvements on Jones and Kelly	Most out-of-bounds reads and writes.	No protection of buffer overflow inside structures or use of freed pointers,	30-125%

Most numbers from WIT's paper. Don't take the numbers too seriously, they've been adjusted to have about the same frame of reference. Protections and limitations also require grains of salt.

Summary: WIT and DieHard

- WIT
  - Static analysis + instrumentation to protect writes.
  - Reasonably practical if everything compiled with WIT.
- DieHard
  - Randomize heap object locations and run multiple copies of the program.
  - Replicas fairly impractical, just an interesting academic idea. (My uneducated opinion, of course.)

References

- Akritidis, P. and Cadar, C. and Raiciu, C. and Costa, M. and Castro, M. *Preventing memory error exploits with WIT*. (Oakland 2008)
- Abadi, M. and Budiu, M. and Erlingsson, U. and Ligatti, J. *Control-flow integrity*. (CCS 2005)
- Berger, E.D. and Zorn, B.G. *DieHard: probabilistic memory safety for unsafe languages*. (SIGPLAN 2006)
- Castro, M. and Costa, M. and Harris, T. *Securing software by enforcing data-flow integrity*. (OSDI 2006)