First-Aid: Surviving and Preventing Memory Management Bugs during Production Runs

Qi Gao, Wenbin Zhang, Yan Tang, and Feng Qin

The Ohio State University

Memory Management Bugs are Severe

- Memory management bugs:
  - Programming errors related to memory management
  - E.g., buffer overflows, dangling pointers, etc.
- Causing severe problems during production runs
  - System hangs or crashes
  - System compromises [US-CERT]
  - Long delays for diagnosing and fixing the bugs [Symantec 2006, Arbaugh 2000]

Desired Features for Handling Memory Bugs at Production Runs?

- Quick recovery
  - Improving availability
- Immune from future errors
  - Covering the time window before official bug fixes
- Safe
  - Not introduce new bugs
- Useful diagnosis reports
  - Assisting offline bug diagnosis
- Low overhead
  - For production runs

Existing Solutions

Category	Examples	Limitations
Oblivion-based	Failure-oblivious computing, reactive immune systems	Unsafe
Redundancy-based	N-version programming, recovery blocks, DieHard, Exterminator	Expensive
Avoidance-based	Rx, Archipelago	Expensive or Non-immune

Our Contributions

- First-Aid: A low-overhead method for surviving and preventing memory bugs
  - Environmental change based failure diagnosis
  - Runtime patches for surviving failures and preventing future errors
- Evaluation with seven real-world applications
  - Fast diagnosis and failure recovery (0.887 sec on average)
  - Effective in preventing bug reoccurrence
  - Low runtime overhead (3.7% on average)
  - Informative bug reports

Outline

- Motivation & Introduction
- First-Aid Overview
- Design and Algorithms
  - Software architecture
  - Diagnosis algorithm
  - Validation algorithm
- Evaluation
- Conclusion

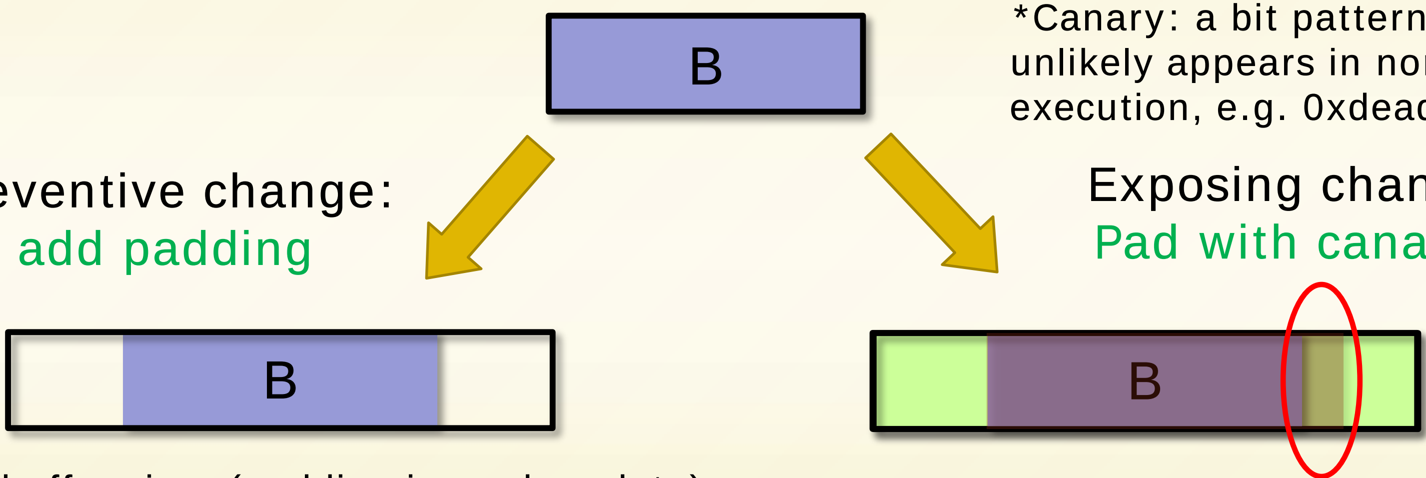
Environmental Changes for Failure Diagnosis

- Two types of environmental changes for diagnosis:
  - Preventive changes
  - Exposing changes

- Execution environments:
  - Everything but the program itself
  - E.g., runtime systems, operating systems, etc.

An Example of Preventive and Exposing Changes

*Canary: a bit pattern that unlikely appears in normal execution, e.g. 0xdeadbeef



Preventive change:
add padding

Exposing change:
Pad with canary*

Enlarge buffer size: (padding is random data)
➔ can prevent failure but not proving occurrence
(possibly cure other types due to disturbance)

1. Detect Overflow!!!
2. Identify bug-affected objects

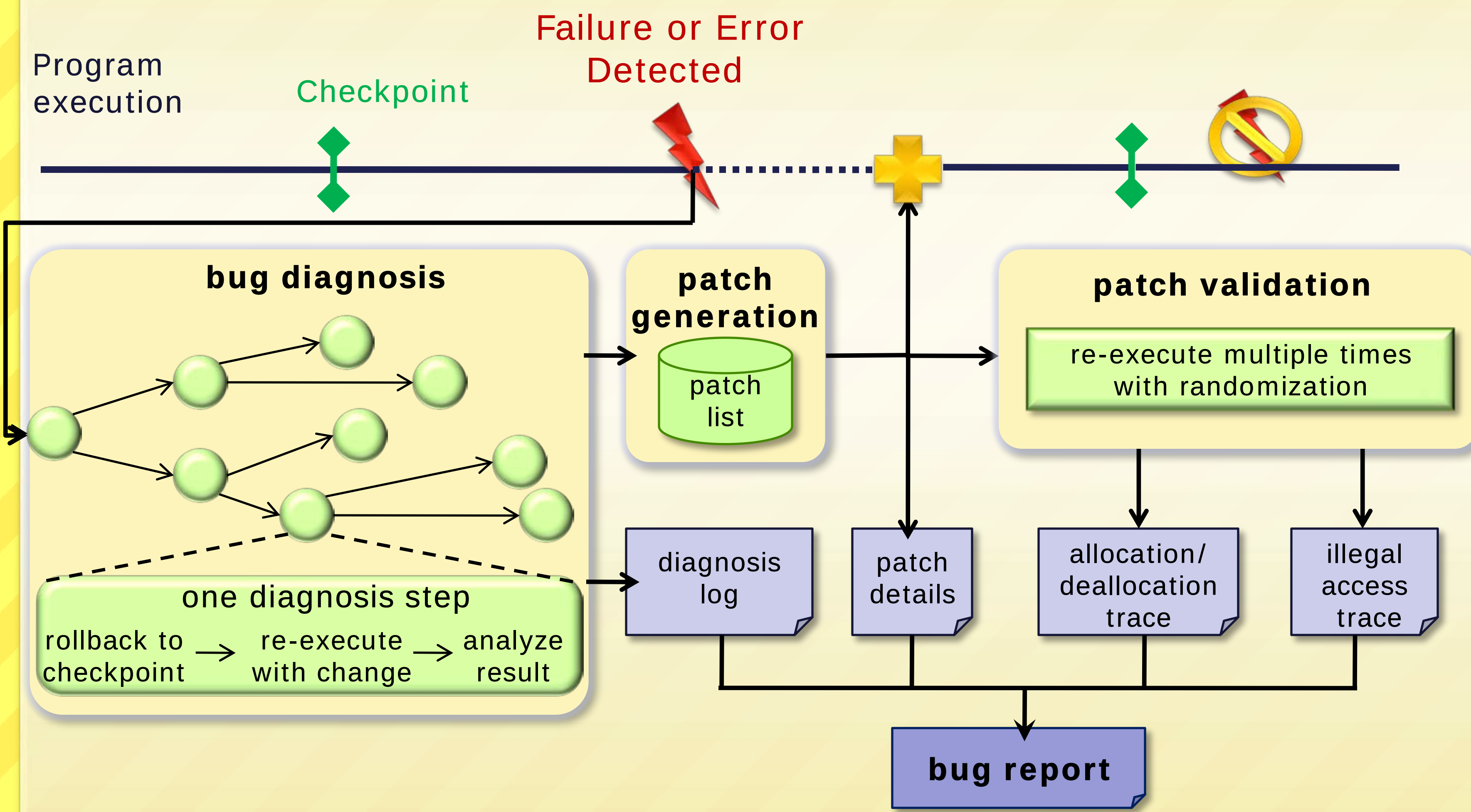
Environmental Changes for Different Types of Memory Bugs

Bug types	Preventive changes	Exposing changes (Bug manifestations)	Application points
Buffer overflow	Padding new objects	Padding objects with canary (corruption)	allocation
Dangling pointer read	Delay free	Fill objects with canary (failure)	deallocation
Dangling pointer write	Delay free	Fill objects with canary (corruption)	deallocation
Double free	Delay free	Check parameters (free twice)	deallocation
Uninitialized read	Fill new objects with zeros	Fill new objects with canary (failure)	allocation

Runtime Patches

Bug types	Preventive changes/ Runtime patches	Exposing changes (Bug manifestations)	Application points
Buffer overflow	Padding new objects	Padding objects with canary (corruption)	allocation
Dangling pointer read	Delay free	Fill objects with canary (failure)	deallocation
Dangling pointer write	Delay free	Fill objects with canary (corruption)	deallocation
Double free	Delay free	Check parameters (free twice)	deallocation
Uninitialized read	Fill new objects with zeros	Fill new objects with canary (failure)	allocation

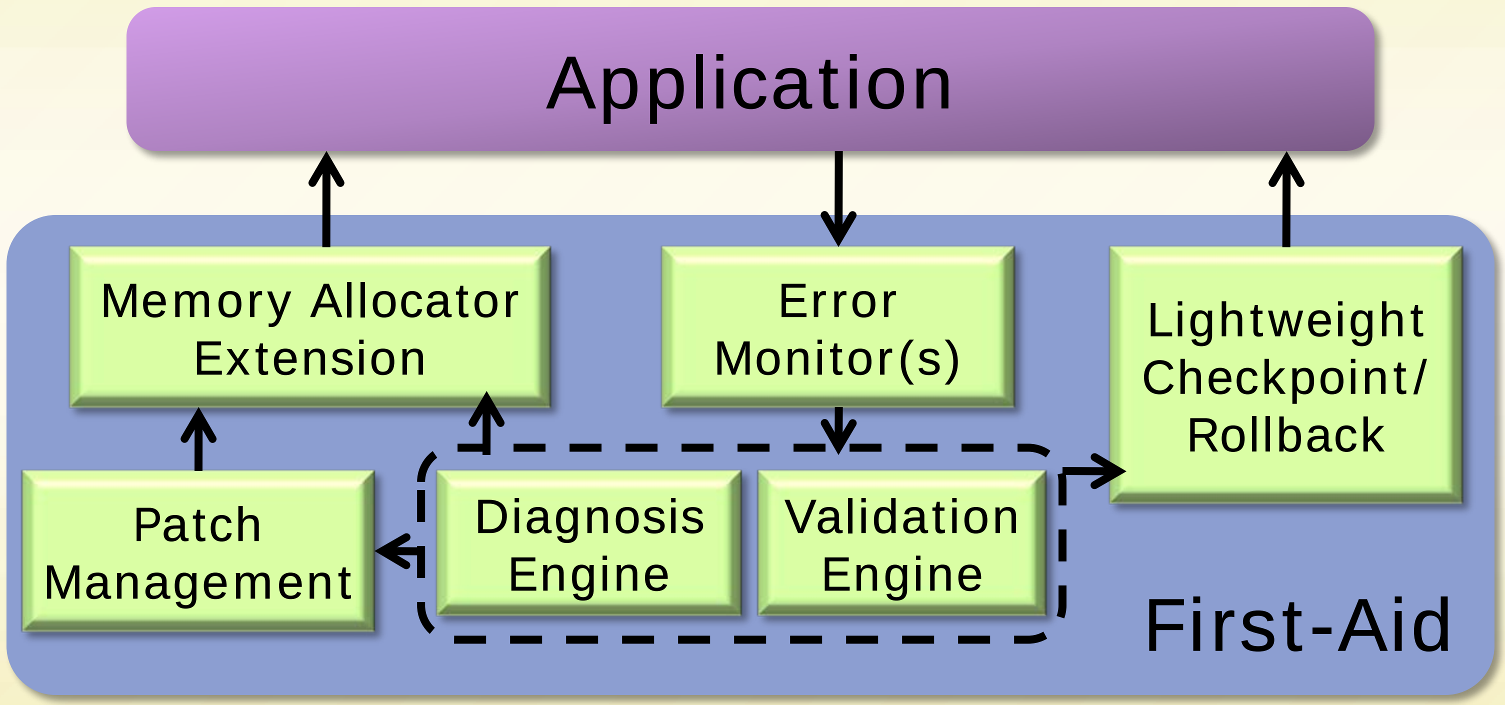
First-Aid Working Scenario



Outline

- Motivation & introduction
- First-Aid overview
- Design and algorithms
  - Software architecture
  - Diagnosis algorithm
  - Validation algorithm
- Evaluation
- Summary

First-Aid Architecture



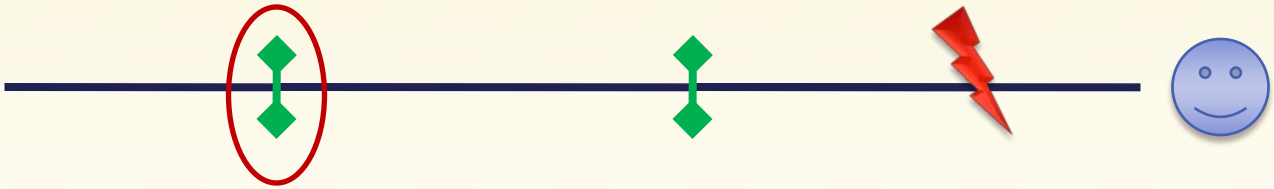
Diagnosis Engine

- Phase I:
  - Is the failure due to memory bug(s)?
  - Which checkpoint to rollback to for diagnosis and patching?
- Phase II:
  - Which type(s) of memory bug(s) has occurred?
  - What memory objects are potentially affected by the bug?

Diagnosis Phase I

Phase I: Is the failure due to memory bug(s)? Which checkpoint to rollback to?

Pass



Re-execute:
All preventive changes
on **All** objects
from this checkpoint

Rollback

We know:

1. A memory bug
2. Triggered after this checkpoint

Diagnosis Phase II

Phase II: Which bug type? Where to patch?

Manifested

Not manifested

Re-execute:
exposing one type, and
preventing other types
on all memory objects

Locate the call-sites by:
1. check corruption, or
2. binary search

Call-site:
[0x806437b]
[0x80651a8]
[0x8074d94]

undecided set

double free

identified set

buffer overflow

We know:
1. Buffer overflow bug
2. Exact call-sites

Enough for patch generation

Validation Engine

Validation: Does the patch have consistent effects?

Randomized allocation

Instrumentation

Iteration 1: allocation/ deallocation trace | illegal access trace

Iteration 2: allocation/ deallocation trace | illegal access trace

Iteration 3: allocation/ deallocation trace | illegal access trace

E.g. read before initialization; write over boundary; etc.

Cross check:
1. patch triggering
2. illegal accesses
3. offset of each illegal access

In parallel with recovered program

Outline

- Motivation & introduction
- First-Aid overview
- Design and algorithms
  - Software architecture
  - Diagnosis algorithm
  - Validation algorithm
- Evaluation
- Summary

Experimental Setup

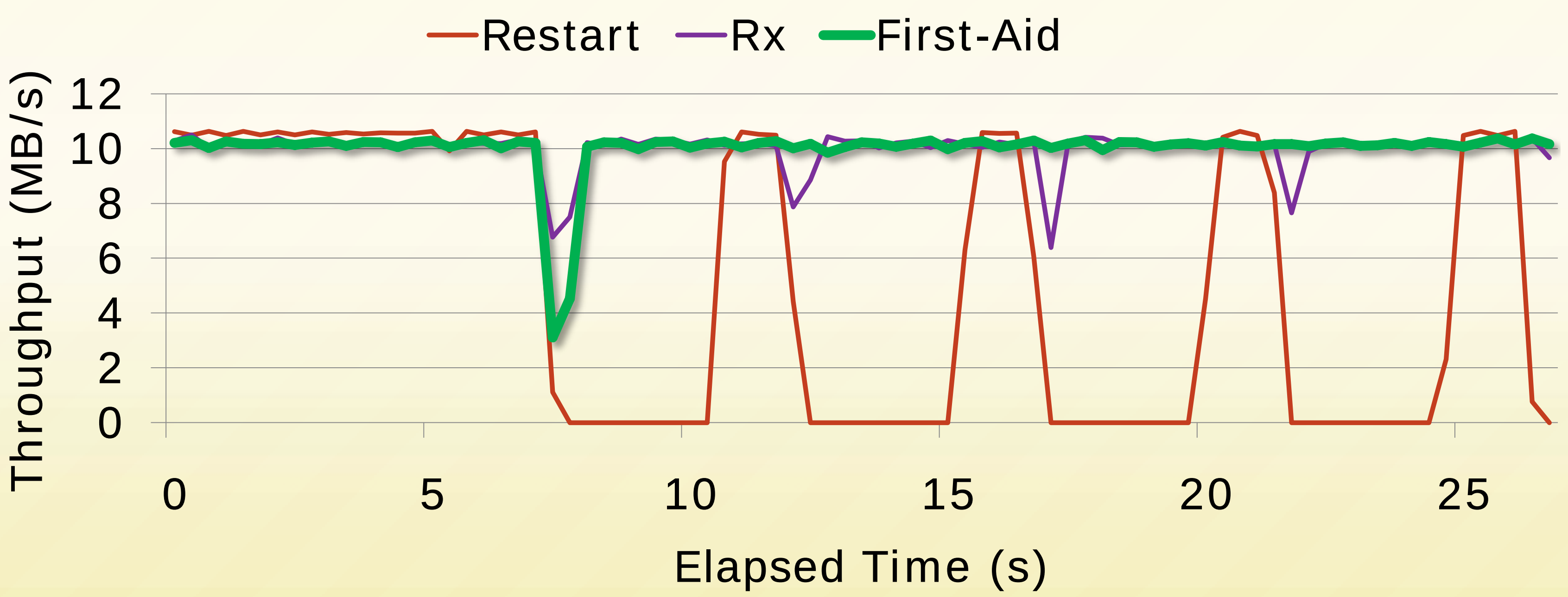
- Implementation:
  - Linux 2.4.22 with flashback checkpointing support
  - Extension based on Lea allocator (used in GNU libc)
- Platform:
  - Intel Xeon 3.00 GHz, 2MB L2 cache, 2GB memory
  - 100 Mbps Ethernet connection
- Applications:
  - Effectiveness: 7 applications (Apache, Squid, CVS, Pine, Mutt, M4, and BC), 7 real bugs, 2 injected bugs
  - Overhead: the above 7 applications, SPEC INT2000, allocation intensive benchmarks

Overall Effectiveness

Application	Diagnosed bugs	Runtime patch (call-sites applied)	Error prevention	Recovery time (s)
Apache	dangling pointer read	delay free (7)	Yes	3.978
Squid	buffer overflow	add padding (1)	Yes	0.386
CVS	double free	delay free (1)	Yes	0.121
Pine	buffer overflow	add padding (1)	Yes	0.722
Mutt	buffer overflow	add padding (1)	Yes	0.617
M4	dangling pointer read	delay free (2)	Yes	1.396
BC	buffer overflow	add padding (3)	Yes	0.573
Apache-uir*	uninitialized read	fill with zero (1)	Yes	0.102
Apache-dpw*	dangling pointer write	delay free (1)	Yes	0.084

Comparison with Rx and Restart

- Trigger the buffer overflow bug in Squid periodically after 7 second

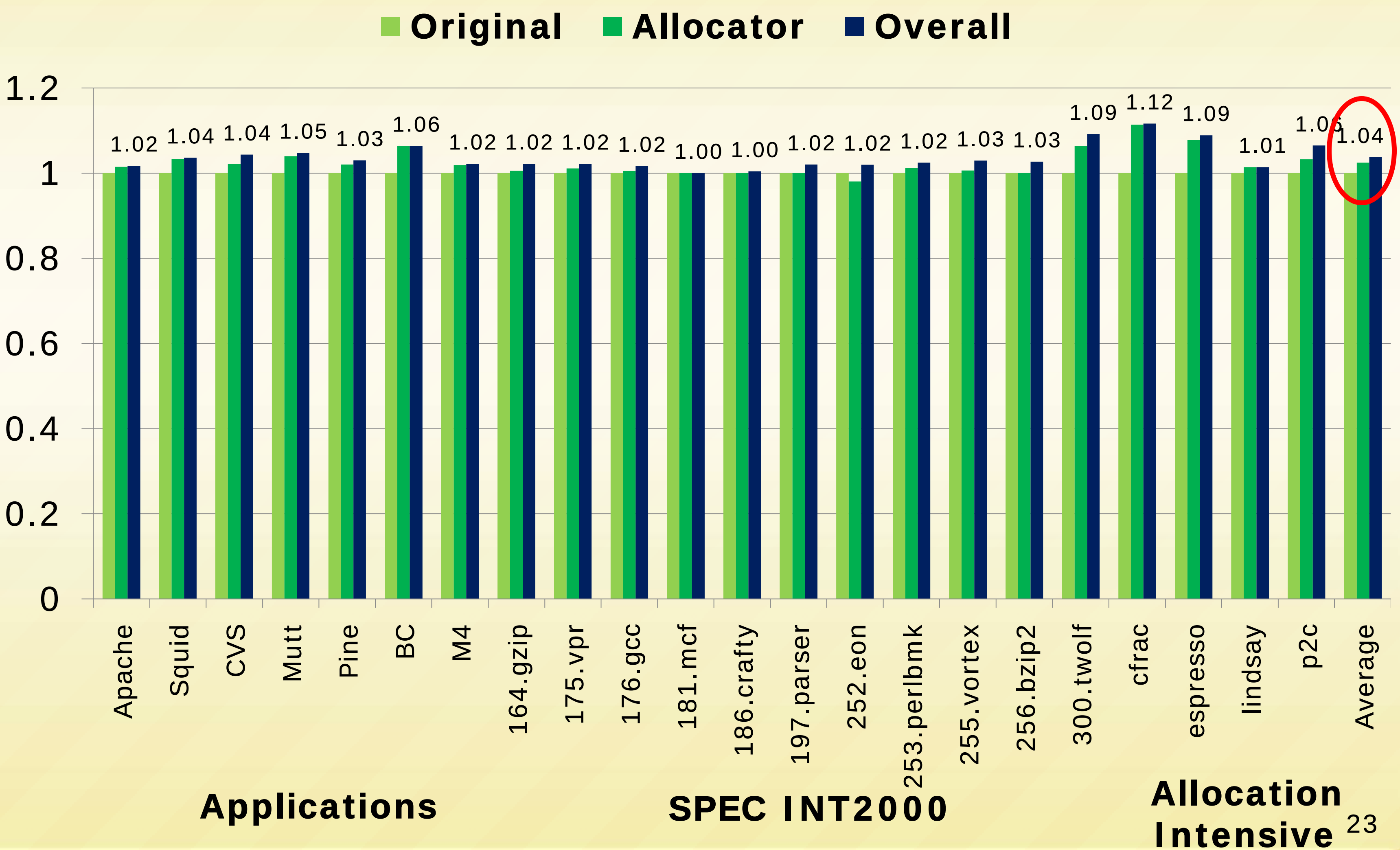


Scope of Patch

- Call-sites and memory objects affected by runtime patches in buggy regions

Name	Call-sites			Objects		
	First-Aid	Rx	Ratio	First-Aid	Rx	Ratio
Apache	7	32	21.88%	315	2567	12.23%
Squid	1	61	1.64%	1	3626	0.03%
CVS	1	44	2.27%	17	306	5.56%
Pine	1	380	0.26%	11	2881	0.38%
Mutt	1	216	0.46%	2	5004	0.04%
M4	2	8	25.00%	3	183	1.64%
BC	3	34	8.82%	5	732	0.68%

Runtime Overhead



Conclusions and Limitations

- Avoidance-based methods with accurate diagnosis can efficiently and effectively survive and prevent memory management bugs.

- Limitations:
  - Cannot handle all types of memory bugs (e.g. memory leaks, incorrect pointer arithmetics)
  - Cannot handle memory bugs that manifest themselves silently
    - Need more powerful error checkers

Future Work and Acknowledgements

- Future Work
  - Evaluate First-Aid with more types of memory bugs in more applications
  - Extend First-Aid to support multi-tier server applications

- Acknowledgements
  - Our shepherd: Julia Lawall
  - Anonymous reviewers
  - Wei Huang, Matthew Koop, Chris Stewart, Guoqing Xu, and Yuanyuan Zhou