FuZZan: Efficient Sanitizer Metadata Design for Fuzzing

Yuseok Jeon[1], WookHyun Han[2], Nathan Burow[1], Mathias Payer[1 3]

hexhive

1 PURDUE UNIVERSITY

2 KAIST

3 EPFL

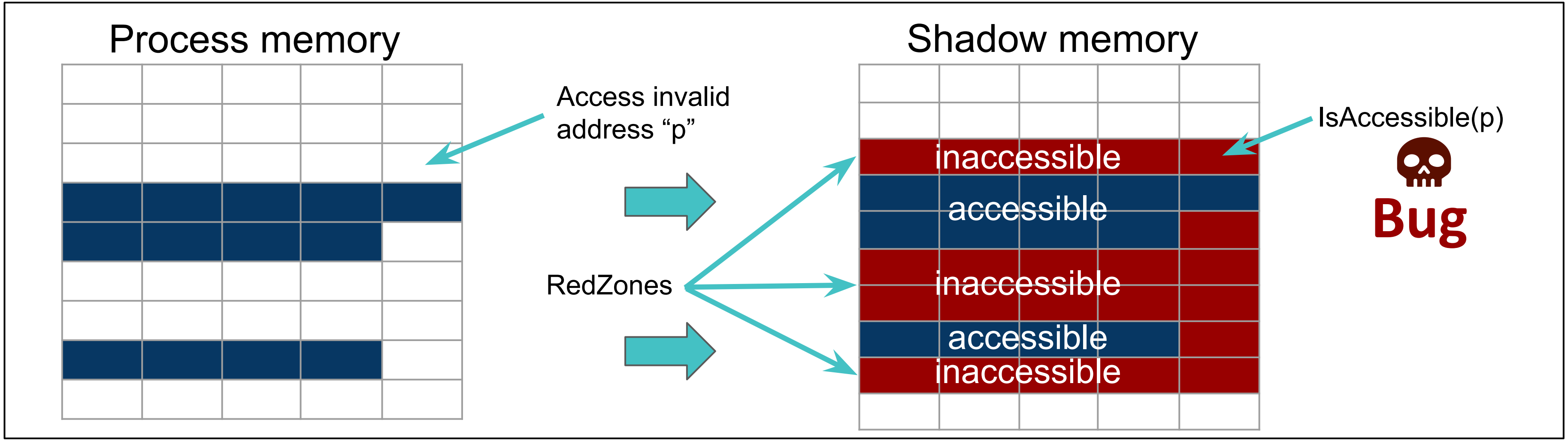
Sanitizer: Debug Policy Violations

- Observe actual execution and flag incorrect behavior
  - E.g., detect memory corruption or memory leak

- Many different sanitizers exist
  - Address Sanitizer (ASan)
  - Memory Sanitizer (MSan)
  - Thread Sanitizer (TSan)
  - Undefined Behavior Sanitizer (UBSan)

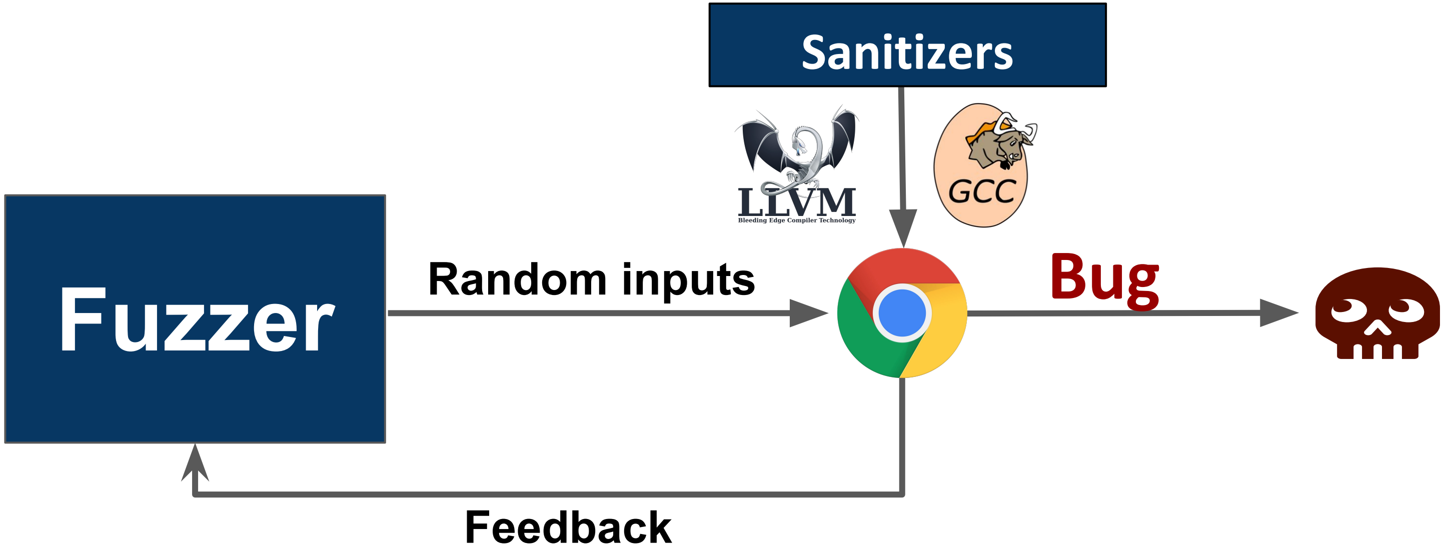
Address Sanitizer (ASan)

- Address Sanitizer is the most widely used sanitizer
  - Focuses on memory safety violations
  - Inserts ***redzone*** around objects
  - Uses ***shadow memory*** to record whether each byte is accessible
  - Detected over 10,000 memory safety violations



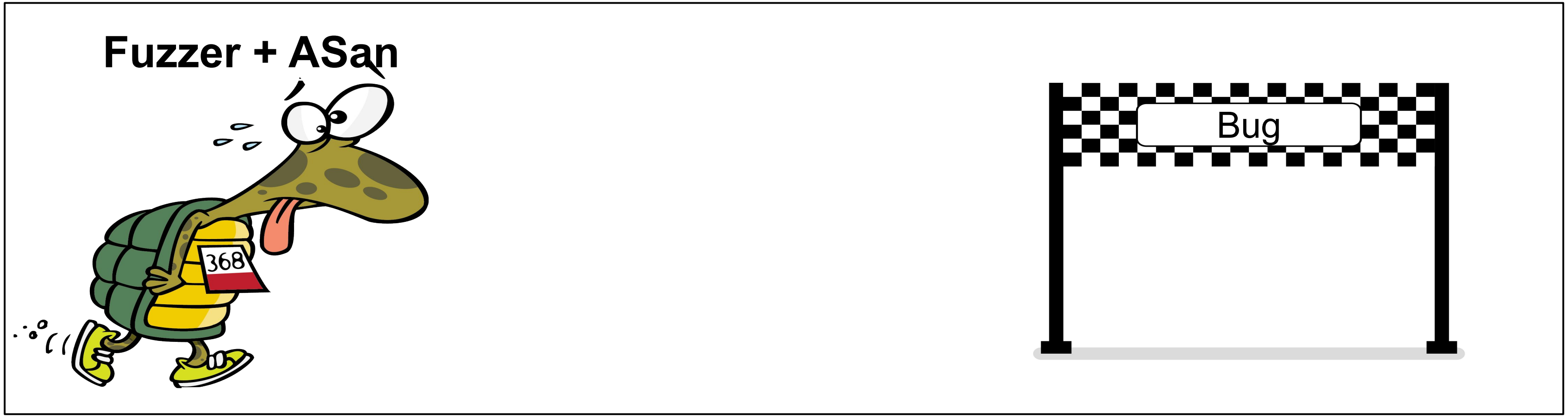
Fuzzing and Context

- Fuzzing is an automated software testing technique
- To detect triggered bugs, fuzzers leverage sanitizers
- Combining a fuzzer with a sanitizer is popular and effective



Motivation

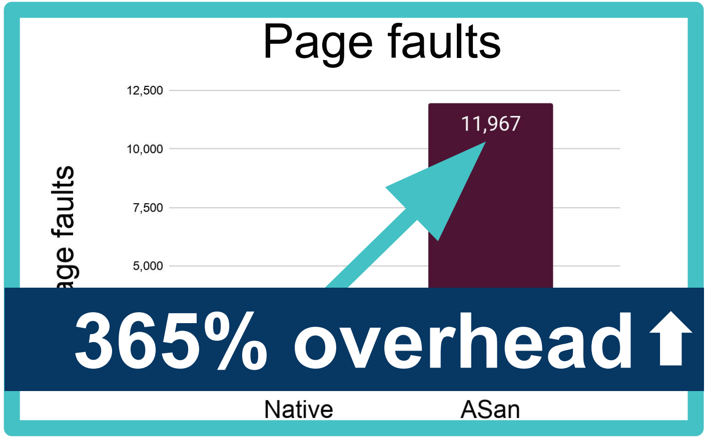
- ❖ Sanitizer is not optimized for fuzzing environment
  - ➢ Highly repetitive and short execution
- ❖ Adapting ASan increases fuzzing performance overhead
  - ➢ E.g., avg 3.4x (up to 6.59x)

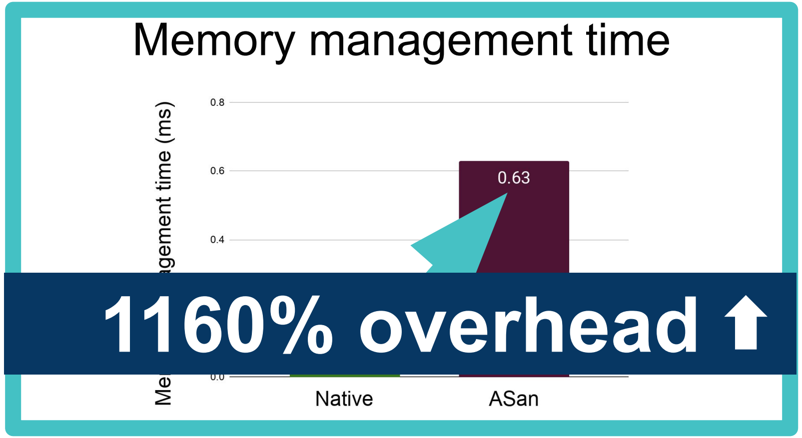


Sanitizers Have High Overhead

(1) Memory management
- ➢ Accessing large virtual memory area incurs overhead
- ➢ Large memory area causes sparse Page Table Entries

(2) ASan initialization
(3) ASan logging





[*] Memory manage functions: (i) do_wp_page, (ii) sys_mmap, (iii) unmap_vmas, and (iv) free_pgtable

FuZZan

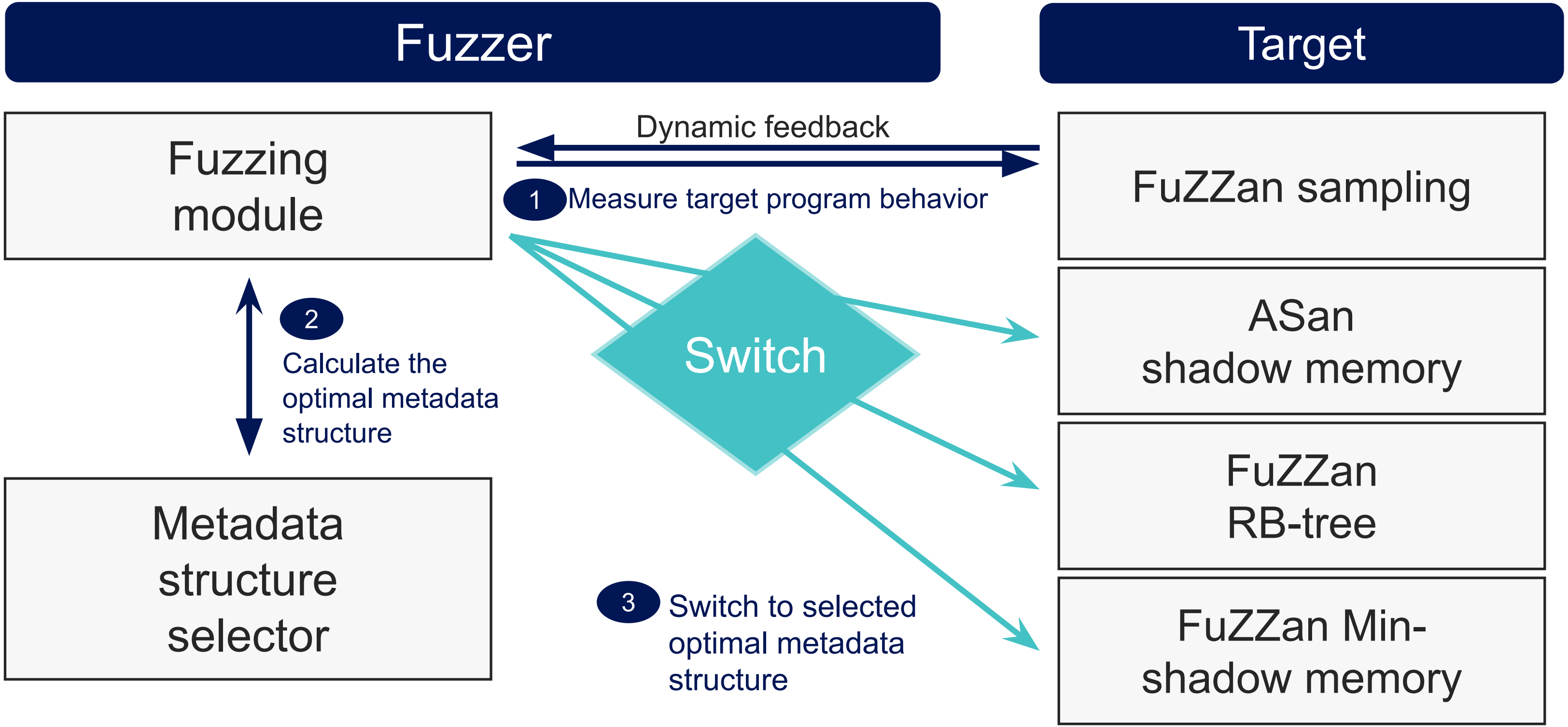
- ❖ Introduce alternate light-weight metadata structures
  - ➢ Avoid sparse Page Table Entries
  - ➢ Minimize memory management overhead

- ❖ Runtime profiling to select optimal metadata structure

- ❖ Remove ASan logging overhead

- ❖ Remove ASan initialization overhead

FuZZan Design

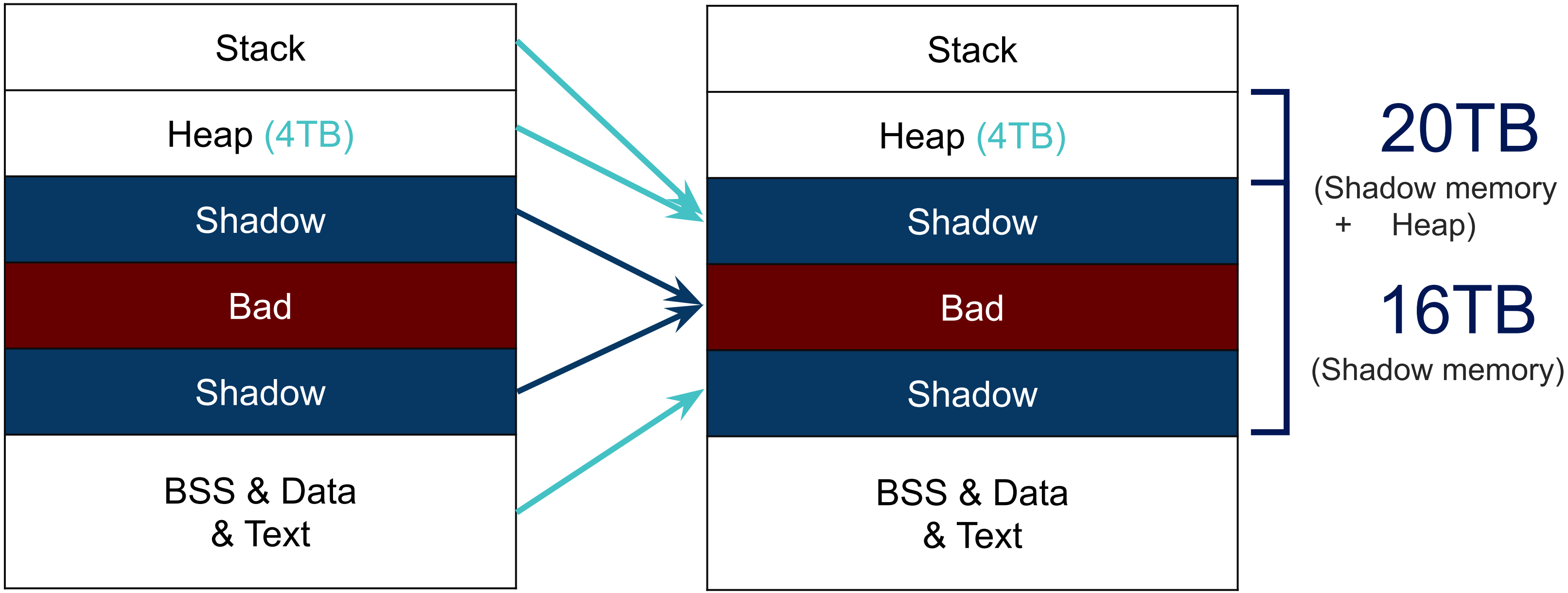


New Metadata Structures

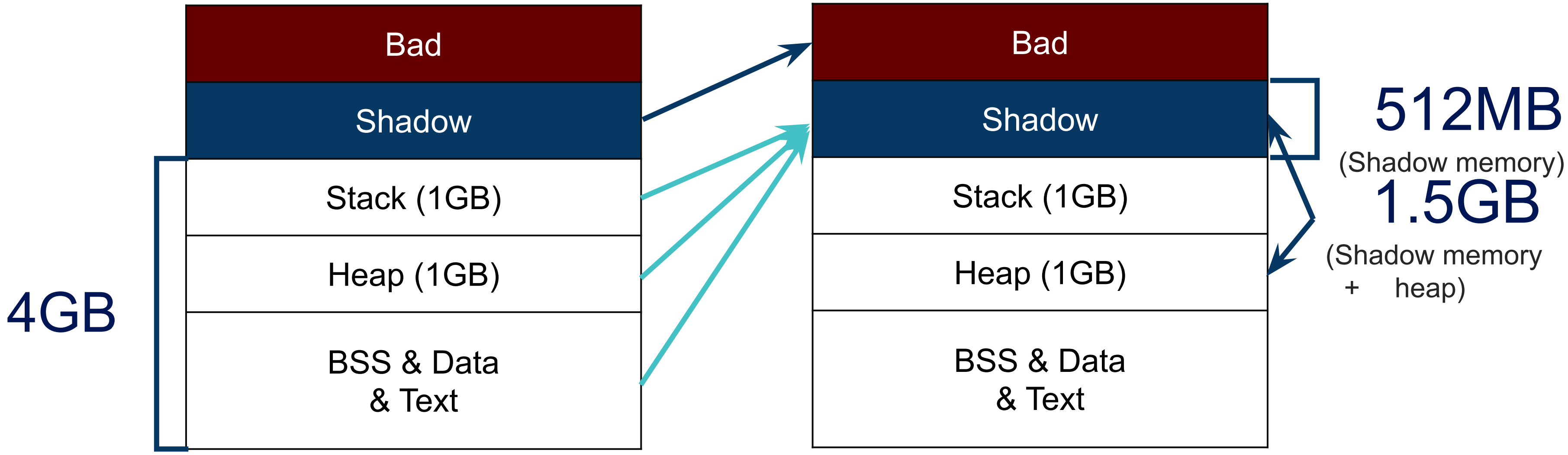
- Propose two different light-weight metadata structures

Metadata Structures		Memory Management Cost	Metadata Access Cost	Target
Address Sanitizer		High	Low O(1)	
FuZZan	RB-tree	Low	High O(log n)	Few metadata access
	Min-shadow	Medium	Low O(1)	Frequent metadata access

ASan Memory Mapping



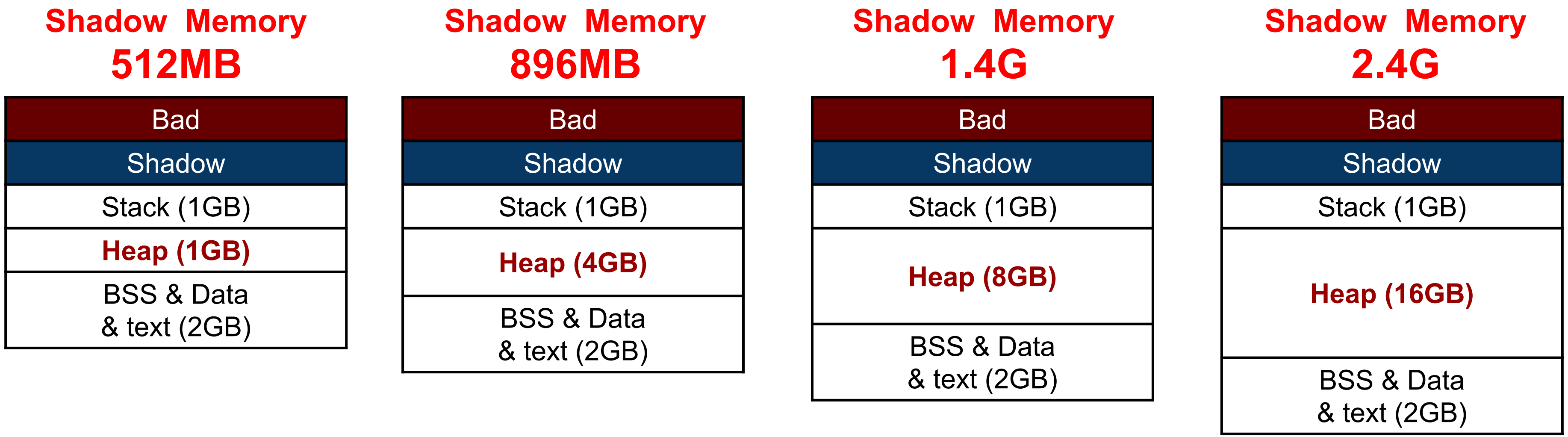
Min-shadow Memory Mapping



20TB -> 1.5GB

Other Min-shadow Memory Modes

- Create additional min-shadow memory modes
  - To accommodate large heap size
  - 1GB, 4GB, 8GB, and 16GB



Dynamic Switching Mode

- ❖ Switch to selected metadata structure during fuzzing

(1) Avoid user's manual extra effort to select optimal metadata structure
  - ➢ No single metadata structure is optimal across all applications
  - ➢ E.g., RB tree for allocating few objects

(2) Change metadata structure according to the target's behavior
  - ➢ Profile at runtime and switch to selected metadata structure
  - ➢ E.g., find new path

(3) Increase heap size when target exceeds limitation

Sampling Mode

- Periodically measure the target program's behavior
  - Metadata access count (stack, heap, and global)
  - Heap object allocation size
- Maintain ASan's error detection capabilities

Initialization/Logging Overhead

- Use *fork server* to avoid unnecessary re-initialization
  - E.g., poisoning of global variable
  - Move ASan’s initialization point before fork server’s entry point
- Modify ASan to disable the logging functionality
  - Complete logging can be recovered with full ASan

Detection Capability

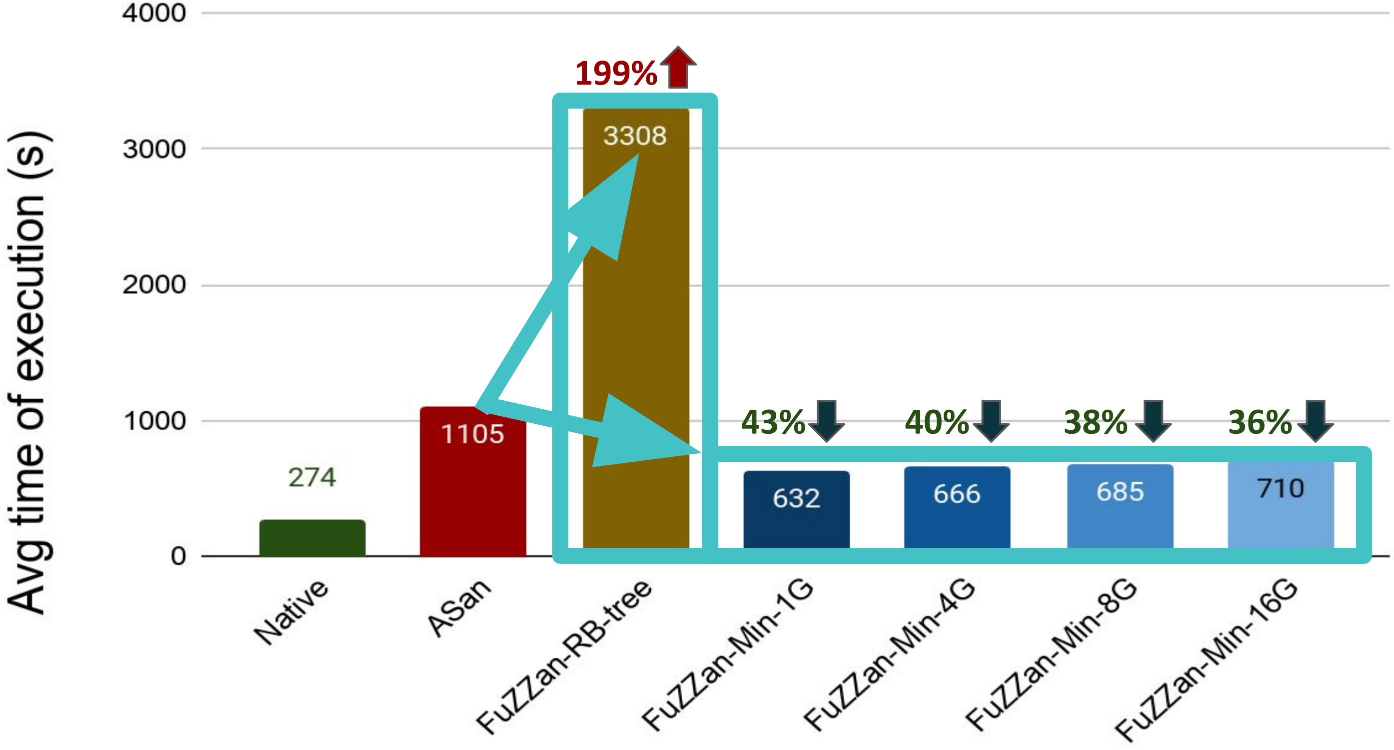
- Juliet Test Suite
  - NIST provides a test suite of all CWEs called Juliet
  - Test using memory corruption CWEs
  - Verified pass or fail all test cases as ASan

- Address Sanitizer provided unit test
  - Verified pass all possible test cases

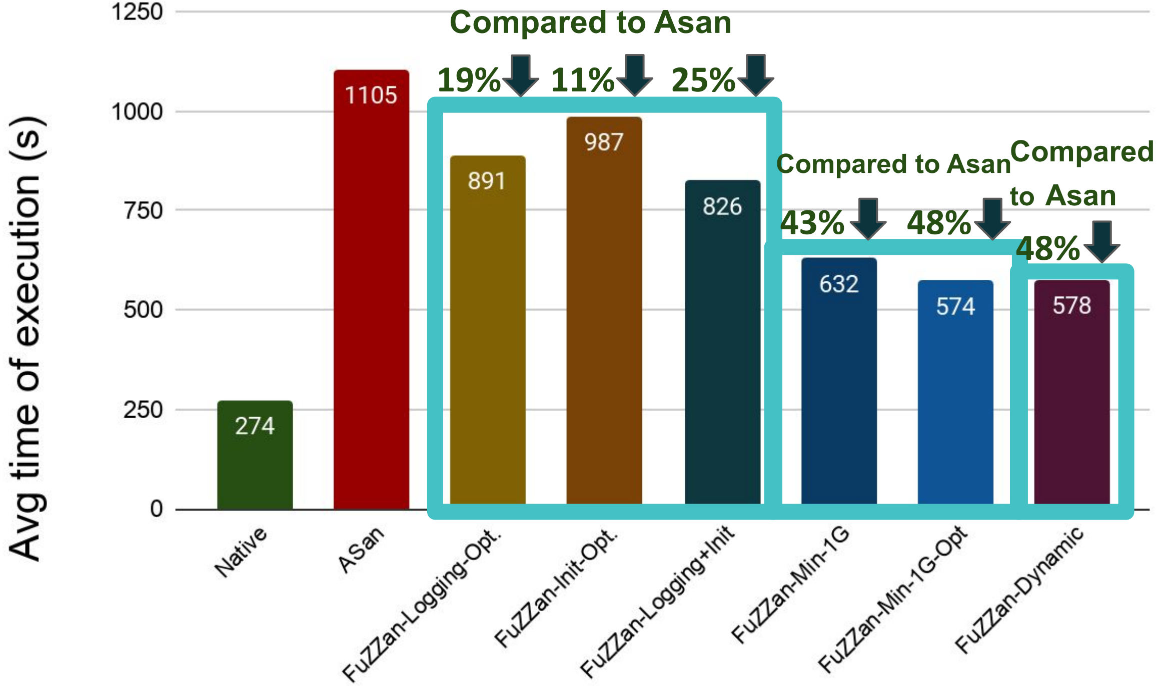
- Fuzzing test using Google Fuzzer Test Suite
  - Fuzzing using 26 applications in test suite
  - Verified same detection capability during fuzzing

CWE: Common Weakness Enumeration

Metadata Structure Performance

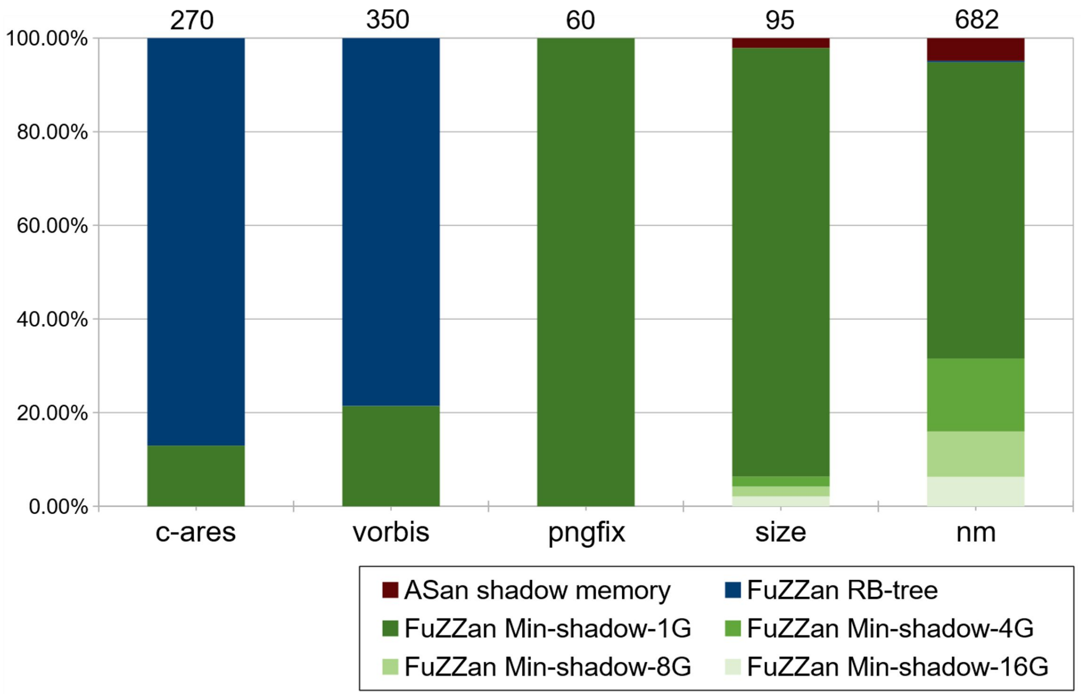


Performance Optimizations



FuZZan-Logging-Opt: optimization for logging overhead
FuZZan-Init-Opt: optimization for Initialization overhead
FuZZan-Min-1G-Opt: min-shadow memory (1G) mode with logging and initialization overhead

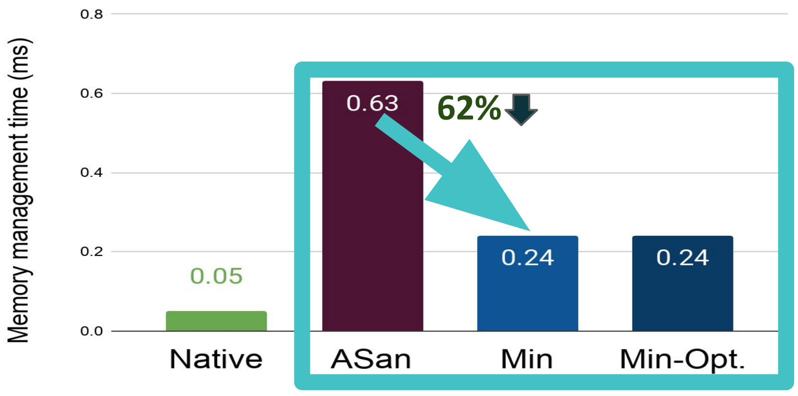
Dynamic Switching Performance



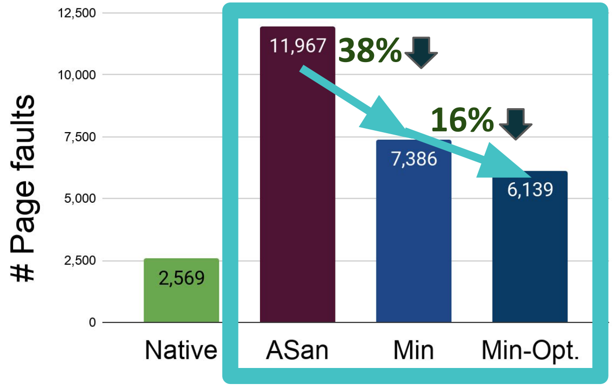
[*] The number on each bar indicates the total metadata switches

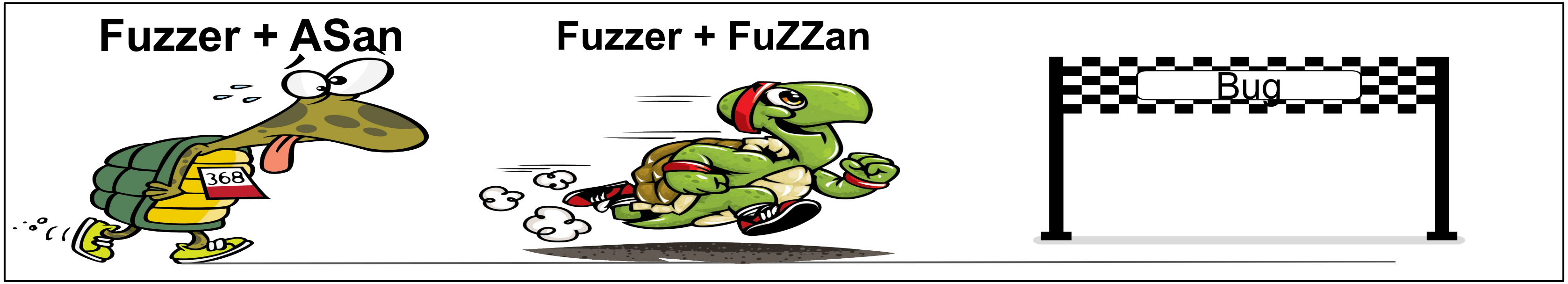
Performance Overhead Analysis

Memory management time

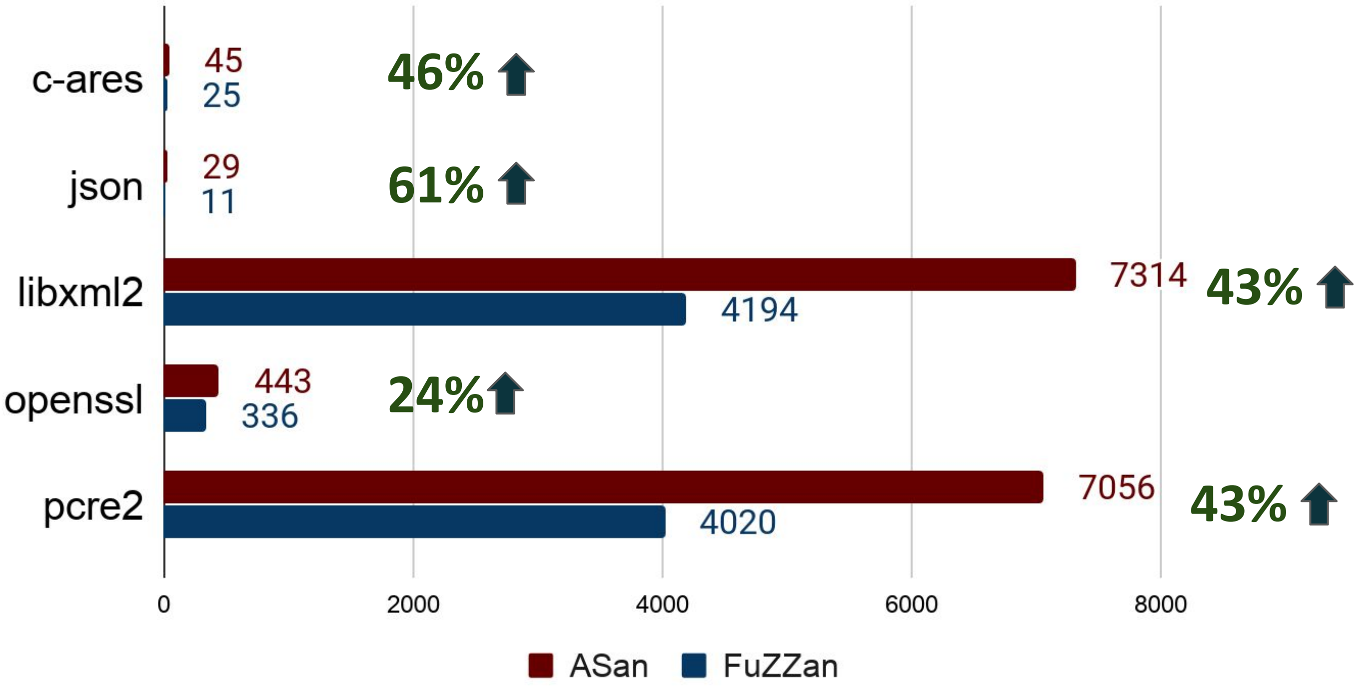


Page faults



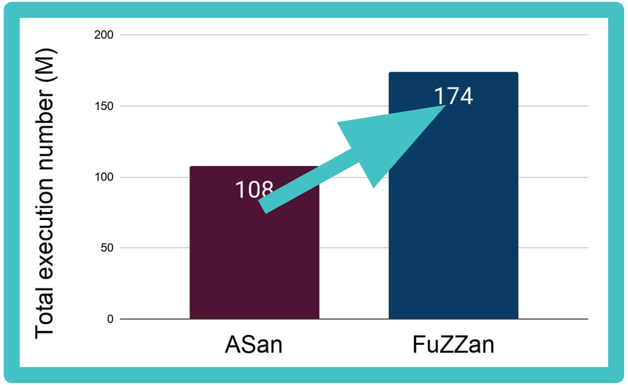


Bug Finding Speed Testing



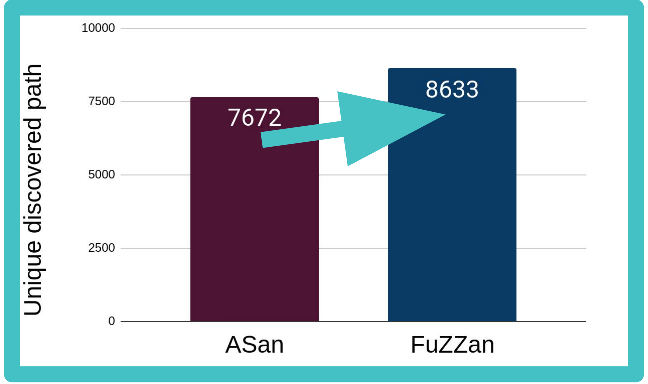
Real-world Fuzz Testing

Total execution number



61% improved

Unique discovered path



13% improved

* the (M) denotes 1,000,000 (one million)

Conclusion

hexhive

- Combining a fuzzer with sanitizer hurts performance

- FuZZan massively reduces performance overhead
  - Novel metadata structures to condense memory space
  - Dynamic switching between metadata structures
  - Removing unnecessary operations

- FuZZan improves fuzzing throughput over ASan
  - Improves fuzzing throughput by 48% starting with provided seeds
    - 52% starting with empty seeds
  - Discovers 13% more unique paths given the same 24 hours
  - Provides flexibility to other sanitizers and AFL-based fuzzers

https://github.com/HexHive/FuZZan