FuZZan: Efficient Sanitizer Metadata Design for Fuzzing

Yuseok Jeon¹, WookHyun Han², Nathan Burow¹, Mathias Payer¹ ³
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)
(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

Sanitizers Have High Overhead

Page faults

Memory management time

365% overhead

1160% overhead
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

Fuzzer

1. Measure target program behavior
2. Calculate the optimal metadata structure
3. Switch to selected optimal metadata structure

Target

1. FuZZan sampling
2. ASan shadow memory
3. FuZZan RB-tree
4. FuZZan Min-shadow memory

Dynamic feedback

Fuzzing module

Metadata structure selector
New Metadata Structures

- Propose two different light-weight metadata structures

<table>
<thead>
<tr>
<th>Metadata Structures</th>
<th>Memory Management Cost</th>
<th>Metadata Access Cost</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Sanitizer</td>
<td>High</td>
<td>Low $O(1)$</td>
<td></td>
</tr>
<tr>
<td>FuZZan</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB-tree</td>
<td>Low</td>
<td>High $O(\log n)$</td>
<td>Few metadata access</td>
</tr>
<tr>
<td>Min-shadow</td>
<td>Medium</td>
<td>Low $O(1)$</td>
<td>Frequent metadata access</td>
</tr>
</tbody>
</table>
ASan Memory Mapping

Stack
- Heap (4TB)
- Shadow
- Bad
- Shadow
- BSS & Data & Text

Stack
- Heap (4TB)
- Shadow
- Bad
- Shadow
- BSS & Data & Text

20TB
(Shadow memory + Heap)

16TB
(Shadow memory)
# Min-shadow Memory Mapping

<table>
<thead>
<tr>
<th>Bad</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Shadow</strong></td>
<td><strong>Shadow</strong></td>
</tr>
<tr>
<td>Stack (1GB)</td>
<td>Stack (1GB)</td>
</tr>
<tr>
<td>Heap (1GB)</td>
<td>Heap (1GB)</td>
</tr>
<tr>
<td>BSS &amp; Data &amp; Text</td>
<td>BSS &amp; Data &amp; Text</td>
</tr>
</tbody>
</table>

- 4GB
- 1.5GB (Shadow memory + heap)
- 20TB -> 1.5GB

512MB (Shadow memory)
Other Min-shadow Memory Modes

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

<table>
<thead>
<tr>
<th>Shadow Memory</th>
<th>512MB</th>
<th>896MB</th>
<th>1.4G</th>
<th>2.4G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad Shadow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack (1GB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap (1GB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSS &amp; Data &amp; text (2GB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shadow Memory</th>
<th>512MB</th>
<th>896MB</th>
<th>1.4G</th>
<th>2.4G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shadow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack (1GB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap (4GB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSS &amp; Data &amp; text (2GB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shadow Memory</th>
<th>512MB</th>
<th>896MB</th>
<th>1.4G</th>
<th>2.4G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shadow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack (1GB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap (8GB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSS &amp; Data &amp; text (2GB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shadow Memory</th>
<th>512MB</th>
<th>896MB</th>
<th>1.4G</th>
<th>2.4G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shadow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack (1GB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap (16GB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSS &amp; Data &amp; text (2GB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

Average time of execution (s)

- Native: 274
- ASan: 1105
- FuZZan-RB-tree: 3308

43% reduction
40% reduction
38% reduction
36% reduction

- FuZZan-Min-1G: 632
- FuZZan-Min-4G: 666
- FuZZan-Min-8G: 685
- FuZZan-Min-16G: 710
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

Compared to Asan
- FuZZan-Logging-Opt: 19% decrease
- FuZZan-Init-Opt: 11% decrease
- FuZZan-Min-1G-Opt: 25% decrease

Compared to Asan
- FuZZan-Logging-Init: 43% decrease
- FuZZan-Min-1G-Opt: 48% decrease
- FuZZan-Dynamic: 48% decrease
Dynamic Switching Performance

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

Memory management time

Page faults

Fuzzer + ASan

Fuzzer + FuZZan
Bug Finding Speed Testing

<table>
<thead>
<tr>
<th>Library</th>
<th>ASan</th>
<th>FuZZan</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>c-ares</td>
<td>45</td>
<td>25</td>
<td>46%</td>
</tr>
<tr>
<td>json</td>
<td>29</td>
<td>11</td>
<td>61%</td>
</tr>
<tr>
<td>libxml2</td>
<td>443</td>
<td>336</td>
<td>24%</td>
</tr>
<tr>
<td>openssl</td>
<td>43</td>
<td>11</td>
<td>43%</td>
</tr>
<tr>
<td>pcre2</td>
<td>4020</td>
<td>4194</td>
<td>43%</td>
</tr>
</tbody>
</table>

7314 43%  
7056 43%  
21
Real-world Fuzz Testing

Total execution number

<table>
<thead>
<tr>
<th></th>
<th>ASan</th>
<th>FuZZan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>108</td>
<td>174</td>
</tr>
</tbody>
</table>

**61% improved**

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

Unique discovered path

<table>
<thead>
<tr>
<th></th>
<th>ASan</th>
<th>FuZZan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>7672</td>
<td>8633</td>
</tr>
</tbody>
</table>

**13% improved**
Conclusion

❖ 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