FuZZan: Efficient Sanitizer Metadata Design for Fuzzing

Yuseok Jeon
Purdue University

Wookhyun Han
KAIST

Nathan Burow
Purdue University

Mathias Payer
EPFL

Abstract

Fuzzing is one of the most popular and effective techniques for finding software bugs. To detect triggered bugs, fuzzers leverage a variety of *sanitizers* in practice. Unfortunately, sanitizers target long running experiments—e.g., developer test suites—not fuzzing, where execution time is highly variable ranging from extremely short to long. Design decisions made for developer test suites introduce high overhead on short lived fuzzing executions, decreasing the fuzzer's throughput and thereby reducing effectiveness.

The root cause of this sanitization overhead is the heavy-weight metadata structure that is optimized for frequent metadata operations over long executions. To address this, we design new metadata structures for sanitizers, and propose FuZZan to *automatically* select the optimal metadata structure without any user configuration. Our new metadata structures have the *same* bug detection capabilities as the ones they replace. We implement and apply these ideas to Address Sanitizer (ASan), which is the most popular sanitizer.

Our evaluation shows that on the Google fuzzer test suite, FuZZan improves fuzzing throughput over ASan by 48% starting with Google's provided seeds (52% when starting with empty seeds on the same applications). Due to this improved throughput, FuZZan discovers 13% more unique paths given the same 24 hours and finds bugs 42% faster. Furthermore, FuZZan catches all bugs ASan does; i.e., we have not traded precision for performance. Our findings show that sanitizer performance overhead is avoidable when metadata structures are designed for fuzzing, and that the performance difference will have a meaningful difference in squashing software bugs.

1 Introduction

Fuzzing [33] is a powerful and widely used software security testing technique that uses randomly generated inputs to find bugs. Fuzzing has seen near ubiquitous adoption in industry, and has discovered countless bugs. For example, the state-of-the-art fuzzer American Fuzzy Lop (AFL) has discovered hundreds of bugs in widely-used software [57], while Google has found 16,000 bugs in Chrome and 11,000 bugs in over 160 other open source projects using fuzzing [10]. On its own, fuzzing only discovers a subset of all triggered bugs, e.g., failed assertions or memory errors causing segmentation faults. Bugs that silently corrupt the program's memory state, without causing a crash, are missed. To detect such bugs, fuzzers must be paired with sanitizers that enforce security policies at runtime by turning a silent corruption into a crash. To date, around 34 sanitizers [47] have been prototyped. So far, only the LLVM-based sanitizers ASan, MSan, LeakSan, UBSan, and TSan have seen wide-spread use. For brevity, we use *sanitizers* to refer to such frequently used sanitizers in the rest of the paper.

Unfortunately, sanitizers are designed for developer-driven software testing rather than fuzzing, and are consequently optimized for minimal per-check cost, not startup/teardown of the metadata structure. Consequently, they are based around a shadow-memory data structure wherein the address space is partitioned, and metadata is encoded into the "shadow" memory at a constant offset from program memory. Optimizing for long executions makes sense in the context of developer-driven software testing, which generally verifies correct behavior on expected input, leading to relatively long test execution times. Fuzzing has a more diverse set of inputs that cause both short (i.e., invalid inputs) and long running executions with billions of executions. For example, the Chrome developers use Address Sanitizer (ASan) for their unit tests and long-running integration tests [39]. However, the underlying design decisions that make ASan a highly performant sanitizer for long running tests result in high performance overhead—up to 6.59×—for short executions, as observed in a fuzzing environment[1]. This high overhead reduces throughput, thereby preventing a fuzzer from finding bugs effectively.

We analyze the source of this overhead across a variety of sanitizers, and attribute the cost to heavy-weight metadata structures employed by these sanitizers. For example, Address Sanitizer maps an additional 20TB of memory for each exe-

[1] The average time for a single execution across the first 500,000 tests for the full Google fuzzer test suite is 0.61ms.

cution, Memory Sanitizer (MSan) 72TB, and Thread Sanitizer (TSan) 97TB on a 64-bit platform. The high setup/teardown cost of heavy-weight metadata structures is amortized over the long execution of programs due to the low per-check cost. In contrast, a fuzzing campaign typically consists of massive amounts of short-lived executions, effectively transforming what is a large one-time cost into a large runtime cost. For example, Table 1 indicates that memory management is the main source of overhead for ASan under fuzzing on the Google fuzz test suite, accounting for 40.16% of the total execution time we observe. Memory management is the key bottleneck for using sanitizers with fuzzers, and has to date gone unaddressed.

Instead, increasing the efficiency and efficacy of fuzzing has received significant research attention on two fronts: (i) mechanisms that reduce the overhead of fuzzers [27, 55, 57]; and (ii) mechanisms that reduce the overhead of sanitization on longer running tests and conflicts between sanitizers [25, 37, 38, 52, 54]. These works address fuzzers and sanitizers in isolation, ignoring the core sanitizer design decision to optimize for long running test cases using a heavy-weight metadata structure that limits sanitizer performance in combination with fuzzers. Consequently, optimization of sanitizer memory management for short execution times remains an open challenge, motivated by the need to design sanitizers that are optimal under fuzz testing.

We present FuZZZan, which uses a two-pronged approach to optimize sanitizers for short execution times, as seen under fuzzing: (i) two new light-weight metadata structures that trade significantly reduced startup/teardown costs [2] for moderately higher (or equivalent) per access costs and (ii) a dynamic metadata structure switching technique, which dynamically selects the optimal metadata structure during a fuzzing campaign based on the current execution profile of the program; i.e., how often the metadata is accessed. Each of our proposed metadata structures is optimized for different execution patterns; i.e., they have different costs for creating an entry when an object is allocated versus looking up information in the metadata table. By observing the metadata access and memory usage patterns at runtime, FuZZZan dynamically switches to the best metadata structure *without* user interaction, and tunes this configuration throughout the fuzzing campaign.

We apply our ideas to ASan, which is the most widely used sanitizer [43, 44, 47]. ASan focuses on memory safety violations—arguably the most dangerous class of bugs, accounting for 70% of vulnerabilities at Microsoft [34]—and has already detected over 10,000 memory safety violations [9, 12, 50] in various applications (e.g., over 3,000 bugs in Chrome in 3 years [50]) and the Linux kernel (e.g., over 1,000 bugs [12, 51]) by using a customized kernel address sanitizer (KASan). We further apply FuZZZan to MSan and MOpt-AFL.

FuZZZan improves fuzzing throughput over ASan by 52% when starting with empty seeds and 48% when starting with Google's seed corpus, averaged across all applications in the Google fuzzer test suite [11] as part of our input record/replay fuzzing experiment. Due to this improved throughput, FuZZZan discovers 13% more unique paths (with an improvement in throughput of 61% compared to ASan) given the standard 24 hour fuzz testing with widely used real-world software and a provided corpus of starting seeds.

Crucially, FuZZZan achieves this without *any* reduction in bug-finding ability. Therefore, FuZZZan strictly increases the performance of ASan-enabled fuzzing, resulting in finding the *same* bugs in *less* time than using ASan with the same fuzzer.

Our contributions are:

1. Identifying and analyzing the primary source of overhead when sanitizers are used with fuzzing, and pinpointing the sanitizer design decisions that cause the overhead;

2. Designing and implementing a sanitizer optimization (FuZZZan) and applying it to ASan; that is, we design several new metadata structures along with a dynamic metadata structure switching to choose the optimal structure at runtime. We also validate the generality of our design by further applying it to MSan and MOpt-AFL;

3. Evaluating FuZZZan on the Google fuzzer test suite and other widely used real-world software and showing that FuZZZan effectively improves fuzzing throughput (and therefore discovers more unique bugs or paths given the same amount of time).

Modes	ASan's init time ms (%)	ASan's logging time ms (%)	Memory mgmt. time ms (%)	# page faults
Native	0.00 (0.00%)	0.00 (0.00%)	0.05 (11.49%)	2,569
ASan	0.17 (10.58%)	0.30 (18.86%)	0.63 (40.16%)	11,967

Table 1: Comparison between native and ASan executions with a breakdown of time spent in memory management, and time spent for ASan's initialization and logging. Results are aggregated over 500,000 executions of the full Google fuzzer test suite [11]. Times are shown in milliseconds, and % denotes the ratio to total execution time.

2 Background and Analysis

We present an overview of fuzzing overhead and ASan (our target sanitizer). Further, we detail the design conflicts between ASan and fuzzing when used in combination.

2.1 Fuzzing overhead

Given the same input generation capabilities, a fuzzer's throughput (executions per second) is critical to its effectiveness in finding bugs. Greater throughput results in more code

[2] Compared to ASan, our min-shadow memory mode reduces the time that startup/teardown functions spend in the kernel by 62% on the first 500,000 tests across the full Google fuzzer test suite.

and data paths being explored, and thus potentially triggers more bugs. Running a fuzzer imposes some overhead on the program, a major component of which is the repeated execution of the target program's initialization routines. These routines—including program loading, `execve`, and initialization—do not change across test cases, and hence result in repeated and unnecessary startup costs. To reduce this overhead, many fuzzers leverage a *fork server*. A fork server loads and executes the target program to a fully-initialized state, and then clones this process to execute each test case. This ensures that the execution of each test case begins from an initialized state, and removes the overhead associated with the initial startup.

Another technique for reducing process initialization costs is *in-process fuzzing*, such as AFL's persistent mode and libFuzzer. In-process fuzzing wraps each test in one iteration of a loop in one process, thus avoiding starting a new process for each test. However, in-process fuzzing generally requires manual analysis and code changes [13, 58]. Additionally, in-process fuzzing requires the target code to be stateless across executions as all tests share one process environment, otherwise the execution of one test may affect subsequent ones, potentially leading to false positives. Consequently, testers should avoid in-process fuzzing for library code using global variables. Bugs found from in-process fuzzing may not be reproducible as it is not always possible to construct a valid calling context to trigger detected bugs in the target function, and side-effects across multiple function calls may not be captured [32]. Because of these limitations, in-process fuzzing is used on stateless functions in libraries, while the fork server model (i.e., out-of-process fuzzing) remains the most general fuzzing mode for fuzzing programs.

2.2 Address Sanitizer

All sanitizers leverage a customized metadata structure [47]. Out of many different metadata schemes, shadow memory (both direct-mapped or multi-level shadow) is the most widely used [4, 14–16, 29, 30, 42, 45, 48, 49, 56]. ASan enforces memory safety by encoding the accessibility of each byte in shadow memory. Allocated (and therefore accessible) areas are marked and padded with inaccessible red zones. In particular, *direct-mapped shadow memory* encodes the validity of the entire virtual memory space, with every 8-bytes of memory mapping to 1-byte in shadow memory. Shadow memory encodes the state of application memory. The 8-bit value k encodes that the 8-k bytes of the mapped memory are accessible. The corresponding shadow memory address for a byte of memory is at:

$$addr_{shadow} = (addr >> 3) + offset$$

where *addr* is the accessed address. Generally, ASan only inserts redzones to the high address side of each object as the preceding object's redzone suffices for the low address side. ASan also instruments each runtime memory access to check if the accessed memory is in a red zone, and if so faults. ASan's effectiveness in detecting hard-to-catch memory bugs has led to its widespread adoption. It has become best practice [47] to use ASan (or KASan [20], the kernel equivalent) with a fuzzer to improve the bug detection capability.

2.3 Overhead Analysis of Fuzzing with ASan

To understand ASan's overhead with fuzzing, we analyze the Linux kernel functions used during fuzzing campaigns. Table 1 shows the overhead added by ASan, broken out across ASan's logging, ASan's initialization, and memory management. Our experiments measure the ratio of the time spent in the kernel functions compared to the total execution time for a number of target programs.

Note that memory management makes up 40.16% of ASan's total execution time, as opposed to 11.49% for the base case, and that memory management is more than double the overhead of ASan's logging and initialization *combined*. ASan's heavy use of the virtual address space results in 4.66× page faults compared to native execution. Our memory management overhead numbers reflect the time spent by the kernel in the four core page table management functions: (i) `unmap_vmas` (24.6%), (ii) `free_pgtable` (4.7%), (iii) `do_wp_page` (8.2%), and (iv) `sys_mmap` (2.6%).

Notably, `unmap_vmas` and `free_pgtable` correspond to 73% of ASan's measured memory management overhead across the four core page table management functions. The execution time for these two functions (`unmap_vmas` and `free_pgtable`) is 10x higher than when executing without ASan. To break this overhead down, when executing a test under the fork server mode, a fuzzer needs to create a new process for each test. During initialization, ASan reserves memory space (20TB total, including 16TB of shadow memory, and a separate 4TB for the heap on 64-bit platforms) and then poisons the shadow memory for globals and the heap. Accessing these pages incurs additional page faults, and thus page table management overhead in the kernel. Note that the large heap area causes sparse page table entries (PTEs), which increase the number of pages used for the page table and memory management overhead.

Existing techniques to deal efficiently with large allocations do not help here. Lazy page allocation of the large virtual memory area used by ASan does not mitigate memory management overhead in this case, as many of the pages are accessed when shadow memory is poisoned. Poisoning forces a copy even for copy-on-write pages, and thus increases page table management cost. During execution, memory allocations and accesses cause additional shadow memory pages to be used, again with page faults and page table management. When the process exits, the kernel clears all page table entries through `unmap_vmas` and releases memory for the page table (via `free_pgtables`). The cost of these two functions are correlated with the number of physical pages used by the process. As fuzzing leads to repeated, short executions, such bookkeeping introduces

considerable memory management overhead. In contrast to these active memory management functions, `sys_mmap` only accounts for 7% memory management overhead of ASan. This is the expense for reserving all virtual memory areas. However, large areas that are actively accessed by ASan incur considerable additional expenses as detailed above.

For completeness, we note that our analysis finds that ASan performs excessive "always-on" logging (18.86%) by default, and that ASan's initial poisoning of global variables (10.58%) is inefficient. Combined, these additional sources of overhead account for 29.44% overhead. We address these engineering shortcomings in our evaluation, but they are neither our core contributions nor the choke point in fuzzing with ASan.

3 FuZZZan design

FuZZZan has two design goals: (1) define new light-weight metadata structures, and (2) automatically switch between metadata structures depending on the runtime execution profile. In this section, we present how we design each component of FuZZZan to achieve both goals, as illustrated in Figure 1.

3.1 FuZZZan Metadata Structures

To minimize startup/teardown costs while maintaining reasonable access costs, FuZZZan introduces two new metadata structures: (i) a Red Black tree (RB-tree) metadata structure, which has low startup and teardown costs, but has high per-access costs; and (ii) min-shadow memory, which has medium startup/teardown costs and low per-access costs (on par with ASan). Table 2 shows a qualitative comparison of the different metadata schemes that we propose in this section, see Table 4 for quantitative results. The RB-tree is optimal for short executions with few metadata accesses as it emphasizes low startup and teardown costs, while min-shadow memory is best suited for executions with a mid-to-high number of metadata accesses as it has lower per metadata access

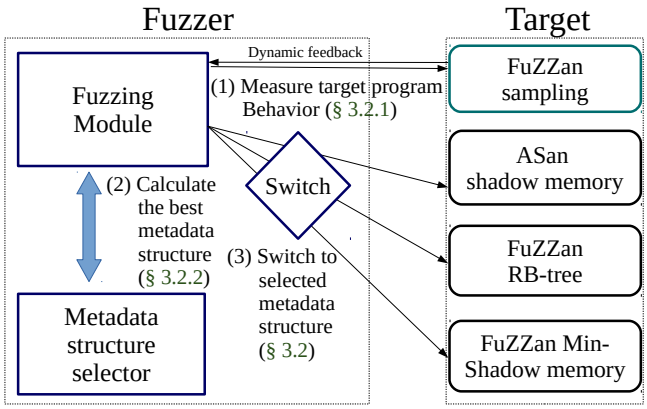

Figure 1: Overview of FuZZZan's architecture and workflow.

Metadata Structures		Startup/ Teardown Cost	Access Cost
ASan shadow memory		High	Low
FuZZZan	Customized RB-tree	Low	High
	Min-shadow memory	Medium	Low

Table 2: Comparison of metadata structures.

costs while still avoiding the full startup/teardown overhead imposed by ASan's shadow memory.

3.1.1 Customized RB-Tree

To optimize ASan's metadata structure for test cases where a fuzz testing application only executes for a very short time with few metadata accesses, we introduce a customized RB-tree, shown in Figure 2. Nodes in the RB-tree store the redzone for each object. Although each metadata access operation (insert, delete, and search) in the RB-tree is slower than its counterpart in the shadow memory metadata structure, our RB-tree has the following benefits: (i) low total memory overhead (leading to low startup/teardown overhead); (ii) removal of poisoning/unpoisoning page faults (as each RB-tree node compactly stores the redzone addresses and these nodes are grouped together in memory); and (iii) a faster range search than shadow memory for operations such as `memcpy`. For example, in order to check `memcpy`, ASan must validate each byte individually using shadow memory. However, in our approach, we can verify such operations through only two range queries for `memcpy`'s source and destination memory address range.

In our RB-tree design, when an object is allocated (e.g., through `malloc`), the range of the object's high address redzone is stored in a node of the RB-tree. During a query, if the address range of the target is lower than the start address of the node, we search the left subtree (and vice versa). If the address is not found in the tree, it is a safe memory access. During redzone removal, the requested address range may only be a subset of an existing node's range (and not the full range of a target node in the RB-tree). In this case, the RB-tree

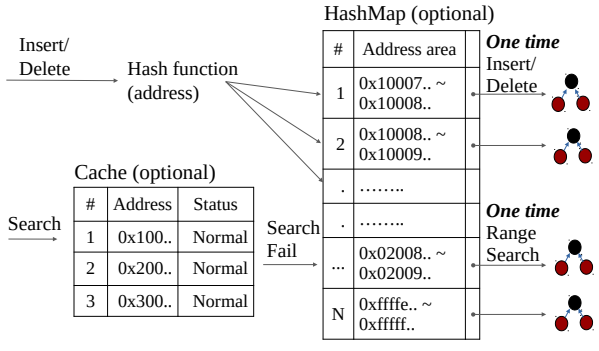

Figure 2: Design of FuZZZan's customized RB-tree.

deletes the existing RB-tree node, creates new RB-tree nodes which have non-overlapping address ranges (e.g., the left and right side of an overlapped area), and inserts these nodes into the RB-tree. Since we reuse ASan's memory allocator and memory layout (e.g., redzones between objects and a quarantine zone for freed objects), FuZZan provides the same detection capability as ASan.

3.1.2 Min-shadow memory

The idea behind Min-shadow memory (for executions with a mid-to-high number of metadata accesses) is to limit the accessible virtual address space, effectively shrinking the size of the required shadow memory. As the size of shadow memory is a key driver of overhead in the fuzzing environment, this enhances performance.

Figure 3 illustrates how min-shadow memory converts a 64-bit program running in a 48-bit address space to run in a 32-bit address space window (1GB for the stack, 1GB for the heap, and 2GB for the BSS, data, and text sections combined). Note that pointers remain 64 bits wide and the code remains unchanged: the mapped address space is simply restricted, allowing min-shadow memory to have a partial shadow memory map. To shrink a program's memory space, we move the heap (by modifying ASan's heap allocator) and remap the stack to a new address space. Min-shadow memory remaps parts of the address space but programs remain 64-bit programs. To accommodate larger heap sizes, we create additional min-shadow

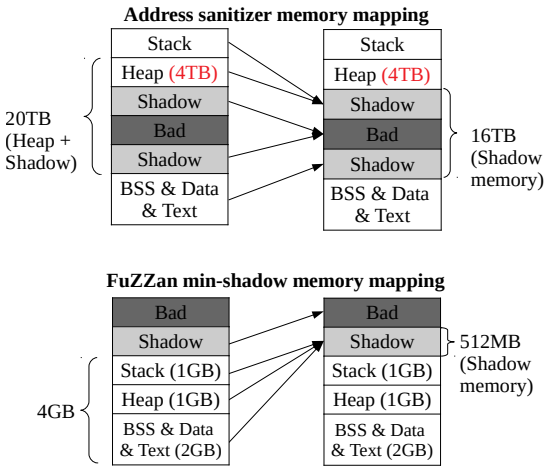

Figure 3: ASan and min-shadow memory modes' memory mapping on 64-bit platforms. ASan (top) reserves 20TB memory space for heap and shadow memory, conversely, min-shadow memory mode (bottom) reserves 4512MB memory space for heap and shadow memory. Each application's stack, heap, and other sections (BSS, data, and text) map to the corresponding shadow regions. Further, the shadow memory region is mapped inaccessible.

memory binaries with heap sizes of 4GB, 8GB, and 16GB.

Our approach allows testing 64-bit code with 64-bit pointers without having to map shadow tables for the entire address space. We disagree with the recommendation of the ASan developers to compile programs as 32-bit executables, as changing the target architecture, pointer length, and data type sizes will hide bugs. Furthermore, min-shadow memory provides greater flexibility compared to using the `x32` ABI [53] mode (i.e., running the processor in 64-bit mode but using 32-bit pointers and arithmetic, limiting the program to a virtual address space of 4GB), as min-shadow memory can provide various heap size options.

3.2 Dynamic metadata structure switching

Dynamic metadata structure switching automatically selects the optimal metadata scheme based on observed behavior. At the beginning of a fuzzing campaign, dynamic metadata structure switching assesses the initial behavior and then periodically samples behavior, adjusting the metadata structure if necessary. Our intuition for dynamic metadata structure switching is that, during fuzzing, metadata access patterns and memory usage remain similar across runs and change in phases. While the fuzzer is mutating a specific input, the executions of the newly created inputs are *similar regarding their control flow and memory access patterns* compared to the source input. However, new coverage may lead to different execution behaviors. We therefore design a *dynamic metadata structure switching* technique that periodically and conditionally samples the execution and adjusts the underlying metadata structure according to the observed execution behavior.

Dynamic metadata structure switching compiles the program in four different ways in preparation for fuzzing: ASan, RB-tree, min-shadow memory, and sampling mode. The sampling mode repeatedly samples the runtime parameters and then selects the optimal metadata structure. The selection of the optimal metadata structure is governed by FuZZan's metadata structure switching policy.

3.2.1 Sampling mode

The sampling mode measures the behavior of the target program using the min-shadow memory-1GB metadata mode and, based on the behavior, reports the currently optimal metadata structure. The sampling mode profiles the following parameters: (i) the number of metadata accesses during insert, delete, and search; and (ii) memory consumption. Note that this information can be collected by simple counters: profiling is therefore light-weight.

Dynamic metadata structure switching starts in sampling mode and selects the optimal mode based on the observed behavior. Dynamic metadata structure switching then periodically (e.g., every 1,000 executions) and conditionally (e.g., when the fuzzer starts mutating a new test case) samples

executions to select the optimal metadata structure based on the current behavior. To reduce the cost of periodic sampling, dynamic metadata structure switching implements a continuous back-off strategy that gradually increases the sampling interval as long as the metadata structure does not change (similar to TCP's slow-start [17]). Note that bugs may be triggered during sampling mode. As such, we maintain ASan's error detection capabilities while sampling to ensure that we do not miss any bugs.

3.2.2 Metadata structure switching policies

Our metadata structure switching policy is based on a mapping of metadata access frequency to the corresponding metadata structure. This heuristic is relatively simple in order to achieve a low sampling overhead. To determine the best cutoff points, we compile all 26 applications in Google's fuzzer test suite in two different ways: RB-tree and min-shadow memory. We then test these different configurations against 50,000 recorded inputs and determine the best metadata structure depending on the observed parameters, measuring execution time. Profiling reveals that the frequency of metadata access (insert, delete, and search) is the primary factor that influences metadata structure overhead, which confirms our original assumption. In this policy, depending on the metadata access frequency, we select different metadata structures (based on statistics from profiling): RB-tree if there are fewer than 1,000 accesses; and min-shadow memory if there are more than 1,000 accesses. Additionally, if the selected heap size goes beyond a threshold, we sequentially switch to other modes (min-shadow memory-4G, 8G, 16G, and ASan), thus increasing heap memory for continuous fuzzing.

4 Implementation

We implement FuZZan's two metadata structures and dynamic metadata structure switching mode on top of ASan in LLVM [28] (version 7.0.0). We support and interact with AFL [57] (version 2.52b). To address the other sources of overhead in ASan (shown in Table 1), we also implement two additional optimizations: (i) removal of unnecessary initialization; and (ii) removal of unnecessary logging. Our implementation consists of 3.5k LOC in total (mostly in LLVM, with minor extensions to AFL).

RB-tree. The RB-tree requires modifications to ASan's memory access instrumentation, as our RB-tree is not based on a shadow memory metadata structure. Thus, we modify all memory access checks, including interceptors, to use the appropriate RB-tree operations instead of the equivalent shadow memory operations. As an optimization, and for compatibility with min-shadow memory mode, the RB-tree mode also reserves 1GB for the heap memory allocator. A compact heap reduces memory management overhead. The RB-tree mode is used when fuzz tests only execute for a very short time with few metadata accesses (i.e., they allocate relatively a small amount of memory).

Min-shadow memory. Unlike the RB-tree, we are able to repurpose ASan's existing memory access checks, as the min-shadow memory metadata structure is based on a shadow memory scheme. To shrink a 64-bit program's address space, we modify ASan's internal heap setup and remap the stack using Kroes et al.'s linker/loader tricks [22]. More specifically, based on this script, we hook `__libc_start_main` using "`LD_PRELOAD`" and then `remap` the stack to a new address, update `rbp` and `rsp`, and then call the original `__libc_start_main`. This allows us to reduce ASan's shadow map requirements from 16TB of mapped (but not necessarily allocated) virtual memory to 512MB (1 bit of shadow for each byte in our 4GB address space window). We also create an additional 192MB shadow memory for ASan's secondary allocator and dynamic libraries (which are remapped above the stack). Finally, we implement four different min-shadow memory modes with increasing heap sizes (1GB, 4GB, 8GB, and 16GB) to handle the different memory requirements of a variety of programs.

Heap size triggers. As previously stated, min-shadow memory is configured for different heap sizes. We therefore use out of memory (OOM) errors to trigger callbacks that notify FuZZan to increase the heap size.

AFL modifications. The target program is compiled once per FuZZan mode. By default, AFL uses a random number generator (RNG) to assign an ID to each basic block within the target program. Unfortunately, this would result in the same input producing different coverage maps across the set of compiled targets, breaking AFL's code coverage analysis. We therefore modify AFL to use the same RNG seed across the set of compiled targets. This ensures that the same input produces the same coverage map across all compiled variants.

Removing unnecessary initialization. ASan makes a number of global constructor calls on program startup, performing several `do_wp_page` calls for copy-on-write. These constructor calls are unnecessarily repeated each time AFL executes a new test input, leading to redundant operations. Unfortunately, the AFL fork server is unaware of ASan's initialization routines. Therefore, to remove unnecessary (re-)initialization across fuzzing runs, we modify ASan's LLVM pass so that global variable initialization occurs *before* AFL's fork server starts. This is achieved by adjusting the priority of global constructors which contain ASan's initialization function.

Removing unnecessary logging. ASan provides logging functionality for error reporting (e.g., saving allocation sizes and thread IDs during object allocation). Unfortunately, this logging functionality introduces additional page faults and performance overhead. However, this logging is unnecessary because fuzzing inherently enables replay by storing test inputs that trigger new behavior. Complete logging information can be recovered by replaying a given input with a

fully-instrumented program. We therefore identify and disable ASan's logging functionality (e.g., `StackDepot`) for fuzzing runs, allowing it to be reenabled for reportable runs.

5 Evaluation

We provide a security and performance evaluation of FuZZZan. First, we verify that FuZZZan and ASan have the same error-detection capabilities. Second, we evaluate the efficiency of FuZZZan's new metadata structures and dynamic metadata structure switching mode using deterministic input from a record/replay infrastructure to ensure fair comparisons. Next, to consider the random nature of fuzzing and to show FuZZZan's real-world impact, we evaluate FuZZZan's efficiency without deterministic input. Here we evaluate the number of code paths found by FuZZZan in a 24 hour time period, demonstrating the impact of FuZZZan's increased performance. We also measure FuZZZan's bug finding speed by using known bugs in Google's fuzzer test suite to verify that FuZZZan maximizes fuzzing execution speed while providing the exact same bug detection capabilities as ASan. Finally, we port FuZZZan to another sanitizer (MSan) [48] and another AFL-based fuzzer (MOpt-AFL) [31] to verify its flexibility.

Evaluation setup. All of our experiments are performed on a desktop running Ubuntu 18.04.3 LTS with a 32-core AMD Ryzen Threadripper 2990WX, 64GB of RAM, 1TB SSD, and Simultaneous MultiThreading (SMT) disabled (to guarantee a single fuzzing instance is assigned to each physical core). Across all experiments, we apply FuZZZan to AFL's fork server mode, which is a widely-used and highly optimized out-of-process fuzzing mode. We evaluate FuZZZan on all applications in the Google fuzzer test suite [11] and other widely used real-world software.

Evaluation strategy. Evaluating fuzzing effectiveness is challenging. In a recent study of how to evaluate fuzzing by Klees et. al. [21], the authors find that the inherent randomness of the fuzzer's input generation can lead to seemingly large but spurious differences in fuzzing effectiveness. However, we are at an advantage as we do not need to compare different fuzzers nor do we change the input generation. We therefore record the fuzzer-generated inputs during a regular run of AFL, and then replay these recorded inputs to compare our different ASan optimizations to the same baseline, effectively controlling for randomness in input generation by using the same input for all experiments. For our experiments we record the first 500,000 executions for replay, yielding a large enough test corpus for reasonable performance comparisons. We also undertake a real-world fuzzing campaign (i.e., without inhibiting fuzzing randomness by record/replay) to measure FuZZZan's real-world impact on code path exploration. Finally, Klees et. al. demonstrate the importance of the initial seed(s) when evaluating fuzz testing, as performance can vary substantially depending on what seed is used. We therefore compare two scenarios: (i) starting with the empty seed; and (ii) starting with a set of valid seeds (we use Google's provided seeds for the input record/replay experiment and randomly selected seeds of the right file type for our real-world fuzz testing).

CWD (ID)	Good tests (Pass/Total)	Bad tests (Pass/Total)
Stack-based Buffer Overflow (121)	2,432 / 2,432	2,314 / 2,432
Heap-based Buffer Overflow (122)	1,594 / 1,594	1,328 / 1,594
Buffer Under-write (124)	682 / 682	641 / 682
Buffer Over-read (126)	524 / 524	359 / 524
Buffer Under-read (127)	682 / 682	641 / 682
Total	5,914 / 5,914	5,283 / 5,914

Table 3: Three different metadata structure modes' detection capability based on the Juliet Test Suite for memory corruption CWEs. FuZZZan and ASan have identical results. Good tests have no memory corruption to check for false positives. Bad tests are intentionally buggy to check for false negatives.

5.1 Detection capability

We verify that FuZZZan and ASan detect the same set of bugs in three different ways. First, we use the NIST Juliet test suite [35], which is a collection of test cases containing common vulnerabilities based on Common Weakness Enumeration (CWE). We use the full Juliet test suite for memory corruption CWEs to verify FuZZZan's capability to detect the same classes of bugs as ASan, without introducing false positives or negatives. Second, to verify that FuZZZan and ASan also have the same detection capability under fuzz testing, we use the Google fuzzer test suite and our recorded input corpus. Finally, we leverage the complete set of ASan's public unit tests as a further sanity check.

For the Juliet test suite (Table 3), we select CWEs related to memory corruption bugs and obtain the same detection results from the three different modes (ASan's shadow memory, RB-tree, and min-shadow memory). To validate FuZZZan against ASan on the Google fuzzer test suite, we compare AFL crash reports across the full set of target programs in the Google fuzzer test suite with our recorded inputs (to identify both false positives and false negatives). Note that we force ASan to crash (the default setting under fuzz testing) when a memory error happens as fuzzers depend on program crashes to detect bugs. As expected, FuZZZan's different modes all obtain the same crash results as ASan. However, we encounter minor differences between FuZZZan and ASan when sanity-checking on the ASan unit tests. These differences are due to internal changes we made when developing FuZZZan, such as min-shadow memory's changed memory layout (failed test cases include features such as fixed memory addresses).

Modes	Empty seed			Provided seed		
	time (s)	vs. Native (%)	vs. ASan (%)	time (s)	vs. Native (%)	vs. ASan (%)
Native	199	-	-	274	-	-
ASan	809	306	-	1,105	303	-
RB-tree	1,541	673	90	3,308	1,106	199
Min-1G	443	122	-45	632	131	-43
Min-4G	465	133	-43	666	143	-40
Min-8G	467	134	-42	685	150	-38
Min-16G	477	139	-41	710	159	-36

Table 4: Comparison between four min-shadow memory modes, RB-tree, Native, and ASan execution overhead during input record and replay fuzz testing with empty and provided seed sets. The time (s) indicates the average of all 26 applications' execution time during testing. Positive percentage (e.g., 20%) denotes overhead while negative percentage indicates a speedup.

5.2 Efficiency of new metadata structures

We perform input record/replay fuzz testing to evaluate the effectiveness of FuZZan's new metadata structures. Doing so isolates the effects of our metadata structures by removing most of the randomness/variation from a typical fuzzing run.

Over the full Google fuzzer test suite, the RB-tree, without any other optimization, shows shorter execution times than ASan if the target application has less than 1,000 metadata accesses; conversely, the RB-tree is slower than ASan when the target application has more than 1,000 metadata accesses. On average, as shown in Table 4, several applications in the Google fuzzer test suite have more than 1,000 metadata accesses, and so RB-tree is overall slower than ASan on average.

Despite being slower on average, the RB-tree can be faster on individual applications and inputs. For instance, FuZZan in RB-tree mode demonstrates a 19% performance improvement (up to 45% faster) for 15 applications (the remaining 11 applications show higher overhead compared to ASan) when benchmarked using the inputs generated from an empty seed. On the subset of applications for which seeds are provided, RB-tree shows less performance improvement (17% and up to 39% faster) for 14 applications (the remaining 12 applications show higher overhead than ASan) when benchmarked using inputs generated from those seeds as provided seeds help to create valid input, lengthening execution times and thus metadata accesses. Note that RB-tree shows the best fuzzing performance when the target application (e.g., `c-ares`) has less 1,000 metadata access. Additionally, even for applications where RB-tree is slower across all inputs, it is still *faster* on inputs with few metadata accesses. The variable performance of RB-tree, which is highly dependent on the number of metadata accesses, highlights the need for dynamic metadata structure switching to automatically select the optimal metadata structure.

Min-shadow memory mode, without additional optimization, outperforms ASan on all 26 programs (for both empty and provided seeds), as shown in Table 4. More specifically, the average improvement is 45% when starting with an empty seed and 43% when starting with the provided seeds. While different min-shadow memory heap configurations show gradual increases in memory overhead (from 1GB to 16GB, in line with the heap size), all of them outperform ASan (at worst, min-shadow memory is still 36% faster than ASan with a provided seed).

Additionally, both metadata configurations can utilize our two engineering optimizations; i.e., removing logging and modifying ASan's initialization (as described in § 4). Table 5 shows that the average improvement of removing unnecessary logging is 24% when starting with an empty seed and 19% when starting with the provided seeds. Similarly, modifying the initialization sequence improves performance by 15% when starting with an empty seed and by 11% when starting with the provided seeds. Combining the two engineering optimizations with min-shadow memory demonstrates synergistic effects: the combined performance is 52% (7% better than native min-shadow memory) faster for empty seeds, and 48% (5% better than native min-shadow memory) faster for provided seeds.

Overall, FuZZan's metadata structures show better perfor-

Modes	Empty seed			Provided seed		
	time (s)	vs. Native (%)	vs. ASan (%)	time (s)	vs. Native (%)	vs. ASan (%)
Logging-Opt.	613	208	-24	891	225	-19
Init-Opt.	686	244	-15	987	260	-11
Logging+Init	552	177	-32	826	201	-25
Min-Shadow	443	122	-45	632	131	-43
Min-Shadow-Opt.	385	93	-52	574	109	-48
Dynamic	387	94	-52	578	111	-48

Table 5: Comparison between FuZZan's three different optimization modes, native min-shadow memory (1G) mode, and min-shadow memory (1G) mode with FuZZan's two optimizations, and dynamic metadata structure switching (Dynamic) mode execution overhead during all 26 applications' input record and replay fuzz testing.

Modes	ASan's init time ms (%)	ASan's logging time ms (%)	Memory manage time ms (%)	Page fault #
Native	0.00 (0.00%)	0.00 (0.00%)	0.05 (11.49%)	2,569
ASan	0.17 (10.58%)	0.30 (18.86%)	0.63 (40.16%)	11,967
Min	0.10 (9.51%)	0.01 (1.33%)	0.24 (24.77%)	7,386
Min-Opt.	0.00 (0.00%)	0.00 (0.00%)	0.24 (24.71%)	6,139

Table 6: Comparison between native, ASan, min-shadow memory (1G), two optimizations with min-shadow memory executions with a breakdown of time spent in memory management, and time spent for ASan's initialization and logging. Results are aggregated over 500,000 executions of the full Google fuzzer test suite. Times are shown in milliseconds, and % denotes the ratio between single execution time and each section execution's time.

mance than ASan's shadow memory for all 26 Google fuzzer test suite applications. As shown in Table 6, the main reasons for FuZZan's improvement are: (i) the smaller memory space reduces memory management overhead as page table management is more lightweight and incurs fewer page faults, (ii) our two engineering optimizations further reduce overhead and number of page faults by removing unnecessary operations, and (iii) the min-shadow memory mode has the same $O(1)$ time complexity for accessing target shadow memory as accessing the original ASan metadata. However, we also observe that the RB-tree is faster than min-shadow memory for some configurations and programs (e.g., `c-ares-CVE`). This motivates the need for dynamic metadata structure switching, which observes program behavior and dynamically selects the best metadata structure based on this behavior.

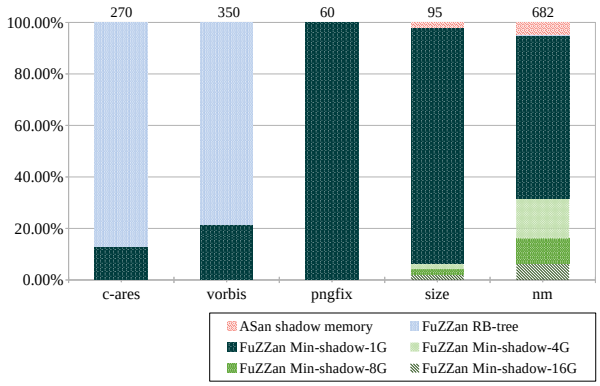

Figure 4: Evaluating the frequency of metadata structure switching and each metadata structure selection over the first 500,000 tests each for c-ares and vorbis in Google's fuzzer test suite and pngfix, size, and nm. The number on each bar indicates the total metadata switches.

5.3 Efficiency of dynamic metadata structure

As described in § 3.2, the dynamic metadata structure switching mode leverages runtime feedback to select the optimal metadata structure, dynamically tuning fuzzing performance according to runtime feedback. The intuition behind the dynamic metadata structure switching mode is that (i) no single metadata structure is best across all applications, (ii) the best metadata structure is not known a priori, so the analyst cannot pre-select the optimal metadata structure, and (iii) fuzzing goes through phases, e.g., alternating between longer running tests (e.g., exploring new coverage) and shorter running tests (e.g., invalid input mutations searching for new code paths). A consequence of the phases of fuzzing is that the same metadata structure is not optimal for every input to a given application. To verify the effectiveness of dynamic metadata structure switching, which is implemented based on these intuitions, we apply dynamic metadata structure switching mode to fuzz testing for seven widely used applications for fuzzing and all 26 applications' in Google's fuzzer test suite.

Our evaluation of dynamic metadata structure switching validates our intuitions, as shown in Figure 4. Observe that different applications are dominated by different metadata structures, e.g., `c-ares` for RB-tree and `pngfix` for min-shadow memory. This is because dynamic metadata structure switching automatically selects the optimal metadata structure (which is unknown a priori). Because dynamic metadata structure switching is automatic, it prevents users from making errors such as selecting RB-tree for applications with a large number of metadata accesses, and removes the need for any user-driven profiling to make metadata decisions. Further, dynamic metadata structure switching scales alongside with the required memory of applications as it increases when the fuzzer finds deeper test cases, as evidenced by `size`, `pngfix`, or `nm` switching to different min-shadow memory modes (4GB, 8GB, and 16GB heap sizes), without user intervention. Without dynamic metadata structure switching, inefficient min-shadow memory modes would be used at the beginning of fuzzing campaigns, or users would have to pause and restart fuzzing campaigns to change metadata modes. As an extreme example highlighting the need for automatic metadata switching, the `nm` benchmark changes metadata structures 682 times, underscoring the infeasibility of having a human analyst determine the single best metadata structure.

As a result of these factors, FuZZan's dynamic metadata structure switching mode improves performance over ASan by 52% when starting with empty seeds and 48% when starting with non-empty seeds. Further, ASan has 306% and FuZZan has 94% (212% less) overhead with empty seeds and ASan has 303% and FuZZan has 111% (192% less) overhead with non-empty seeds compared to native execution. Note that dynamic metadata structure switching has identical fuzzing performance to using min-shadow memory with 1GB heap alone, and improves performance over RB-tree up to 870%. Consequently, automating metadata selection is not adding noticeable overhead, while substantially improving user experience. We recommend using dynamic metadata structure switching mode for the following four reasons: (i) if the target application exceeds FuZZan's heap memory limit (1GB), dynamic metadata structure switching automatically increases the heap size for the few executions that require it (a fixed heap size results in *false positive crashes* due to heap memory exhaustion), (ii) preventing users from selecting an incorrect metadata structure, (iii) using only one metadata structure (e.g., min-shadow memory) may miss the opportunity to further improve throughput, as, in some cases, RB-tree (or some future metadata structure) may be faster than min-shadow memory; (iv) manually selecting a metadata structure requires extra effort (e.g., measuring each metadata structure's efficiency for the target application), which dynamic metadata structure switching mode avoids by automatically selecting the optimal metadata structure.

Programs	Native		ASan		FuZZan	
	exec #	path #	exec #	path #	exec # (%)	path # (%)
cxxfilt	86M	2,769	33M	2,442	51M (55%)	2,651 (9%)
file	29M	1,126	7M	763	9M (29%)	845 (11%)
nm	51M	1,272	7M	822	12M (71%)	872 (6%)
objdump	95M	883	15M	567	17M (13%)	595 (5%)
pngfix	36M	971	18M	912	33M (83%)	982 (8%)
size	52M	703	17M	626	32M (88%)	656 (5%)
tcpdump	70M	3,587	11M	1,540	20M (82%)	2,032 (32%)
Total	419M	11,311	108M	7,672	174M (61%)	8,633 (13%)

Table 7: Evaluating FuZZan's total execution number and unique discovered path for 24 hours fuzz testing with provided seeds. The (M) denotes 1,000,000 (one million) and ratio (%) is the ratio between ASan and FuZZan.

Programs	ASan TTE (s)	FuZZan		Type (source)
		TTE (s)	rate (%)	
c-ares	45	25	46	BO (ares_create_query.c:196)
json	29	11	61	AF (fuzzer-parse_json.cpp:50)
libxml2	7,314	4,194	43	BO (CVE-2015-8317)
openssl-1.0.1f	443	336	24	BO (t1_lib.c:2586)
pcre2	7,056	4,020	43	BO (pcre2_match.c:5968)
Total	14,887	8,586	42	-

Table 8: Evaluating FuZZan's bug finding speed. The TTE denotes the mean time-to-exposure. The AF is assertion error and the BO denotes buffer overflow.

5.4 Real-world fuzz testing

Our experiments validating FuZZan use a record/replay approach to avoid any impact of randomness, allowing meaningful comparisons to a baseline. However, real-world fuzzing is highly stochastic, and so we also evaluate FuZZan in the context of several real-world end-to-end fuzzing campaigns without deterministic input record/replay. For this experiment, we select the following widely used programs: `cxxfilt`, `nm`, `objdump`, `size` (all from binutil-2.31), `file` (version 5.35), `pngfix` (from libpng 1.6.38) and `tcpdump` (version 4.10.0). Klees et al. [21] select and test `cxxfilt`, `nm`, and `objdump` in their fuzzing evaluation study. The remaining four programs (`size`, `file`, `pngfix`, and `tcpdump`) are widely tested by recent fuzzing works [1, 3, 6, 26, 36, 46]. For each binary, we run a fuzzing campaign. Each campaign is conducted for 24 hours and repeated five times. We measure the number of total executions and discovered unique paths when fuzzing with seeds from the seed corpus of each program with the right type file and three different configurations: native, ASan, and FuZZan's dynamic metadata structure switching mode, and report the mean over the five campaigns.

As a result, FuZZan improves throughput over ASan by 61% (up to 88%). Interestingly, FuZZan discovers 13% more unique paths given the same 24 hours time due to improved throughput. Our evaluation also shows that improved throughput increases the possibility of finding more bugs in the same amount of time, as we discuss next.

Modes	time (s)	vs. Native (%)	vs. MSan (%)	vs. MSan nolock (%)
Native	146	-	-	-
MSan	14,074	9,575	-	-
MSan-nolock	386	165	-97	-
Min-16G	335	130	-98	-13

Table 9: Comparison between Native, MSan, MSan-nolock, and min-shadow memory execution overhead during input record and replay fuzz testing with provided seed sets. MSan-nolock disables lock/unlock for MSan's logging depots. Time (s) indicates the average of execution time. Positive percentages denote overhead, negative percentages denote speedup.

5.5 Bug finding effectiveness

FuZZan increases throughput while maintaining ASan's bug detection capability, potentially enabling it to find more bugs. To demonstrate this, we evaluate FuZZan's bug finding speed and compare it to a fuzzing campaign with ASan. In this evaluation, we target five applications in Google's fuzzer test suite. These applications are chosen because we found bugs in them (using ASan and dynamic metadata structure switching mode) within a 24 hour fuzzing campaign. We use the seeds provided by the test suite and repeated each campaign five times. Note that we do not replay recorded inputs during these campaigns, instead letting the fuzzer generate random inputs. Table 8 shows the mean time (over five campaigns) to find each bug. Notably, FuZZan finds all bugs up to 61% (mean 42%) faster than ASan, and is faster in all cases. This experiment emphasizes our belief that throughput is paramount when fuzzing with sanitizers.

5.6 FuZZan Flexibility

Appling FuZZan to Memory Sanitizer. Like ASan, numerous sanitizers use shadow memory for their metadata structure [47]. For example, other popular sanitizers, such as Memory Sanitizer (MSan) [48] and Thread Sanitizer (TSan) [42], also rely on shadow memory for metadata. FuZZan optimizes sanitizer usage of shadow memory *without* modifying the stored shadow information or how the sanitizer uses that information. Consequently, porting our shadow metadata improvements in FuZZan from ASan to other sanitizers is a simple engineering exercise. To demonstrate this, we port FuZZan to MSan. In so doing, we shrink MSan's memory space to implement min-shadow memory 16G for MSan (1GB for the stack, 16GB for the heap, and 2GB for the BSS, data, and text sections combined). We only implement one metadata mode for our MSan proof-of-concept to validate our claim that applies FuZZan to other shadow memory based sanitizers is an engineering exercise.

Table 9 summarizes MSan's performance overhead on different modes for all 26 evaluated applications. Initially,

min-shadow memory shows high overhead—around 96 times native. Analyzing this, we found that MSan's `fork()` interceptor locks all logging depots before `fork()` and similarly unlocks them afterwards to avoid deadlocks. However, as explained in § 4, locking/unlocking logging depots is unnecessary for fuzzing because these logging depots exist for bug reporting and fuzzing inherently enables replay by storing test inputs when the fuzzer finds bugs. We thus disable these lock/unlock functions to create the MSan-nolock mode, which has reasonable overhead (2.6 times that of native).

FuZZan's MSan min-shadow memory 16G mode shows 13% performance improvement compared to MSan-nolock mode, demonstrating FuZZan's efficacy when applied to MSan. We expect that additional optimization and the application of the dynamic switch mode will lead to even higher performance improvement. We leave this engineering as future work.

Applying FuZZan to MOpt-AFL. FuZZan is not coupled to a particular fuzzer or fuzzer version. Most modern fuzzers [2, 3, 31, 31] extend AFL, so our approach applies broadly. To demonstrate this, we apply FuZZan to MOpt-AFL [31], which is an efficient mutation scheduling scheme to achieve better fuzzing efficiency. We modify MOpt-AFL to add FuZZan's profiling feedback and dynamic metadata switching functions. To measure FuZZan's impact on MOpt-AFL, we select seven real-world applications (the same set as Table 7) and fuzz them for 24 hours each, repeating the experiment five times to control for randomness in the results. On average, ASan-MOpt-AFL mode discovers 85% more unique paths given the same 24 hours time due to MOpt-AFL's effectiveness compared to ASan. Notably, FuZZan-MOpt-AFL mode discovers 112% more unique paths (27% higher than ASan-MOpt-AFL) due to the improved throughput.

6 Discussion

In this section, we summarize some potential areas for future work, a possible security extension enabled by FuZZan, and lessons learned in designing FuZZan.

Removing conflicts between sanitizers. ASan's shadow memory scheme conflicts with other sanitizers that are also based on shadow memory, e.g., MSan and TSan. Each sanitizer interprets the shadow memory in a mutually exclusive manner, prohibiting the use of multiple concurrent sanitizers. For example, ASan uses shadow memory as a metadata store, while MSan prohibits access to the same memory range. FuZZan's new metadata structures can be adapted to avoid this conflict, and enable true composition of sanitizers, since we use lightweight, independent metadata structures. Each sanitizer can map its own instance of our metadata structure, and all sanitizers may coexist in a single process. However, some engineering effort is required to port sanitizers to our new metadata structures. An alternate approach would be to have one metadata structure that stores information for all sanitizers. Whether having a unified metadata structure or a metadata structure per sanitizer is more efficient is an interesting research question.

Possible security extension. Unfortunately, ASan's virtual memory requirements directly conflict with fuzzers' abilities to detect certain out-of-memory (OOM) bugs. For example, fuzzers typically limit memory usage to detect OOM errors when parsing malformed input. However, ASan's large virtual memory requirement masks OOM bugs, leaving them undetected because of the difficulty of setting precise memory limits. Consequently, using a compact metadata structure with ASan not only improves performance, but also can enable an extension of ASan's policy to cover OOM bugs.

Lessons Learned. Our initial metadata design leveraged a two-layered shadow memory metadata structure that split metadata lookups into two parts: a lookup into a top-level metadata structure, followed by a lookup into a second-level metadata structure a la page tables. While this design vastly reduced memory consumption and management overhead, the additional runtime cost per metadata access of the additional indirection resulted in the two-layer structure being slower than ASan in all cases.

For dynamic metadata structure switching, we evaluated two additional policies: (i) utilizing more detailed metadata access information such as each object type's (e.g, stack) metadata access (e.g., insert) count and each operation's microbenchmark results, and (ii) running each metadata mode, measuring their execution time, and selecting the fastest metadata mode. In our evaluation, the additional sampling complexity of these policies outweighed any gains from more precisely selecting a metadata structure.

7 Related Work

7.1 Reducing Fuzzing Overhead

Several approaches reduce the overhead of fuzzing. One approach is to reduce the execution time of each iteration. AFL supports a deferred fork server which requires a manual call to the fork server. The analyst is encouraged to use the deferred fork server, and manually initiate the fork server as late as possible to reduce, not only overhead from linking and libc initializations, but also overhead from the initialization of the target program. Deferred mode, however, cannot reduce the teardown overhead of heavy metadata structures. AFL's persistent mode and libFuzzer eliminate the overhead from creating a new process. However, these approaches require manual effort, and users must know the target programs. Xu et al. [55] implement several new OS primitives to improve the efficiency of fuzzing on multicore platforms. Especially, by supporting a new system call, `snapshot` instead of `fork`, they reduce the overhead of creating a process. Moreover, they reduce the overhead from file system contention through a dual file system service.

However, this approach requires kernel modifications for the new primitives, and does not reduce the overhead of sanitizers.

Another approach is to improve fuzzing itself so that it can find more crashes within the same amount of executions. AFLFast [3] adopts a Markov chain model to select a seed. If inputs mutated from a seed explore more new paths, the seed has higher probability to be selected. With given target source locations, AFLGo [2] selects a seed that has higher probabilities to reach the source locations. Several approaches adopt hybrid fuzzing, taint analysis, and machine learning to help fuzzers explore more paths. SAVIOR [8] uses hybrid fuzzing, combining it with concolic execution to explore code blocks guarded by complex branch conditions. RedQueen [1] uses taint analysis and symbolic execution for the same purpose. VUzzer [40] also uses dynamic taint analysis and mutates bytes which are related to target branch conditions to efficiently explore paths. TIFF [18] infers the type of the input bytes through dynamic taint analysis and uses the type information to mutate the input. Matryoshka [7] uses both data flow and control flow information to explore nested branches. In addition to hybrid fuzzing with traditional techniques such as symbolic and concolic executions, NEUZZ [46] adapts neural network and sets the number of covered paths as an objective function to maximize covered paths. Angora [6] adapts both taint analysis and a gradient descent algorithm to improve the number of covered paths. These approaches do not reduce the execution time of each iteration. They are therefore orthogonal to our work. Thus, we can use these approaches to further increase fuzzing performance.

7.2 Optimizing Sanitizers

Since C/C++ programming languages are memory and type unsafe languages, several sanitizers [47] target memory safety violations [5, 23, 41, 48, 49] and type safety violations [14, 19, 24, 29]. Despite their broad use, sanitizers have several limitations such as high overhead, limited detection abilities, and incompatibility with other sanitizers.

To reduce sanitizer overhead, ASAP [52] and PartiSan [25] disable check instrumentation on the hot path according to their policies. The intuition of both approaches is that most of the sanitizer's overhead comes from checks on a few hot code paths that are frequently executed (e.g., instrumentation in a loop). ASAP removes check instrumentation on the hot path based on pre-calculated profiling results at compile time. In PartiSan [25], Lettner et al., propose runtime partitioning to more effectively remove check instrumentation based on runtime information during execution. However, both approaches miss a main source of overhead when reducing the cost of ASan during fuzzing campaigns: the overhead is due to memory management and not due to the low overhead safety checks. As ASAP and PartiSan target the cost of checks, they are complementary to FuZZan. To fuzz quickly, there is an option to generate a corpus from a normal binary, and then feed the corpus to an ASan binary. FuZZan can also adopt this option for fast fuzzing.

Pina et al., [38] use multi-version execution to concurrently run sanitizer-protected processes together with native processes, synchronizing all versions at the system-call level. To synchronize all versions, they use a system-call buffer and a Domain-Specific Language [37] to resolve conflicts between different program versions. Xu et al., [54] propose Bunshin to reduce the overhead of sanitizers and conflicts based on the N-version system through their check distribution, sanitizer distribution, and cost distribution policies. Since these approaches are based on N-version systems, they increase hardware requirements such as several dedicated cores and at least N times of memory. Also, these approaches do not address the fundamental problem of ASan memory overhead.

8 Conclusion

Combining a fuzzer with sanitizers is a popular and effective approach to maximize bug finding efficacy. However, several design choices of current sanitizers hinder fuzzing effectiveness, increasing the runtime cost and reducing the benefit of combining fuzzing and sanitization.

We show that the root cause of this overhead is the heavy metadata structure used by sanitizers, and propose FuZZan to optimize sanitizer metadata structures for fuzzing. We implement and apply these ideas to ASan. We design new metadata structures to replace ASan's rigid shadow memory, reducing the memory management overhead while maintaining the same error detection capabilities. Our dynamic metadata structure adaptively selects the most efficient metadata structure for the current fuzzing campaign without manual configuration.

Our evaluation shows that FuZZan improves performance over ASan 52% when starting with empty seeds (48% with Google's seed corpus). Based on improved throughput, FuZZan discovers 13% more unique paths given the same 24 hours and finds bugs 42% faster. The open-source version of FuZZan is available at https://github.com/HexHive/FuZZan.

Acknowledgments

We thank the anonymous reviewers and our shepherd Julia Lawall for their detailed feedback. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), NSF CNS-1801601, and ONR award N00014-18-1-2674. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[4] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2011.

[5] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. CUP: Comprehensive User-Space Protection for C/C++. In *Proceedings of the Asia Conference on Computer and Communications Security (ASIACCS)*, 2018.

[6] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2018.

[7] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: Fuzzing Deeply Nested Branches. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[8] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Long Lu, et al. SAVIOR: Towards Bug-Driven Hybrid Testing. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2020.

[9] Google. Address Sanitizer Found Bugs. https://github.com/google/sanitizers/wiki/AddressSanitizerFoundBugs.

[10] Google. Clusterfuzz. https://google.github.io/clusterfuzz/.

[11] Google. Fuzzer test suite. https://github.com/google/fuzzer-test-suite.

[12] Google. Kernel Address Sanitizer (KASan), a fast memory error detector for the Linux kernel. https://github.com/google/kasan/wiki.

[13] Google. Libfuzzer tutorial. https://github.com/google/fuzzer-test-suite/blob/master/tutorial/libFuzzerTutorial.md.

[14] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. TypeSan: Practical type confusion detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[15] Niranjan Hasabnis, Ashish Misra, and R Sekar. Light-weight bounds checking. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2012.

[16] Reed Hastings. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Security Symposium (SEC)*, 1992.

[17] Van Jacobson. Congestion avoidance and control. *ACM SIGCOMM computer communication review*, 1988.

[18] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. TIFF: Using Input Type Inference To Improve Fuzzing. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2018.

[19] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. HexType: Efficient Detection of Type Confusion Errors for C++. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[20] Linux kernel document. The Kernel Address Sanitizer (KASAN). https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html.

[21] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[22] Taddeus Kroes, Koen Koning, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Fast and generic metadata management with mid-fat pointers. In *Proceedings of the European Workshop on Systems Security (EuroSec)*, 2017.

[23] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[24] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type Casting Verification: Stopping an Emerging Attack Vector. In *Proceedings of the USENIX Security Symposium (SEC)*, 2015.

[25] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. PartiSan: fast and flexible sanitization via run-time partitioning. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2018.

[26] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*, 2017.

[27] LLVM. LibFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[28] LLVM. The LLVM Compiler Infrastructure Project. http://llvm.org/.

[29] LLVM. TySan: A type sanitizer. https://reviews.llvm.org/D32199.

[30] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Debugging via run-time type checking. In *Processings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2001.

[31] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proceedings of the USENIX Security Symposium (SEC)*, 2019.

[32] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[33] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 1990.

[34] Matt Miller. Trends, challenge, and shifts in software vulnerability mitigation. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

[35] NIST. Juliet test suite. https://samate.nist.gov/SARD/testsuite.php.

[36] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2018.

[37] Luís Pina, Daniel Grumberg, Anastasios Andronidis, and Cristian Cadar. A DSL approach to reconcile equivalent divergent program executions. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[38] Luís Pina, Anastasios Andronidis, and Cristian Cadar. FreeDA: Deploying Incompatible Stock Dynamic Analyses in Production via Multi-Version Execution. In *Proceedings of the ACM International Conference on Computing Frontiers (CF)*, 2018.

[39] The Chromium Project. Address Sanitizer (ASan). https://www.chromium.org/developers/testing/addresssanitizer.

[40] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012.

[42] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications (WBIA)*, 2009.

[43] Kostya Serebryany. Hardware Memory Tagging to make C/C++ memory safe(r). https://github.com/google/sanitizers/blob/master/hwaddress-sanitizer/HardwareMemoryTaggingtomakeC_C++memorysafe(r)-iSecCon2018.pdf.

[44] Kostya Serebryany. Sanitize, Fuzz, and Harden Your C++ Code. https://www.usenix.org/sites/default/files/conference/protected-files/enigma_slides_serebryany.pdf.

[45] Julian Seward and Nicholas Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2005.

[46] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2019.

[47] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: sanitizing for security. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2019.

[48] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[49] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the European Conference on Computer Systems (EUROSYS)*, 2017.

[50] Dmitry Vyukov. Address/Thread/MemorySanitizer Slaughtering C++ bugs. https://www.slideshare.net/sermp/sanitizer-cppcon-russia.

[51] Dmitry Vyukov. Syzbot. https://syzkaller.appspot.com/upstream.

[52] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2015.

[53] Wikipedia. x32 ABI. https://en.wikipedia.org/wiki/X32_ABI.

[54] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. Bunshin: Compositing Security Mechanisms through Diversification. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[55] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[56] Yves Younan. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[57] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl.

[58] Michal Zalewski. New in AFL: persistent mode. https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html.