Unleashing Coveraged-Based Fuzzing Through Comprehensive, Efficient, and Faithful Exploitable-Bug Exposing

Bowen Wang, Kangjie Lu, Qiushi Wu, and Aditya Pakki

Abstract—Fuzzing has become an essential means of finding software bugs. Bug finding through fuzzing requires two parts—exploring code paths to reach bugs and exposing bugs when they are reached. Existing fuzzing research has primarily focused on improving code coverage but not on exposing bugs. Sanitizers such as AddressSanitizer (ASAN) and MemorySanitizer (MSAN) have been the dominating tools for exposing bugs. However, sanitizer-based bug exposing has the following limitations. (1) sanitizers are not compatible with each other. (2) sanitizers incur significant runtime overhead. (3) sanitizers may generate false positives, and (4) exposed bugs may not be exploitable. To address these limitations, we propose Expozzer, a fuzzing system that can expose bugs comprehensively, efficiently, and faithfully. The intuition of Expozzer is to detect bugs through divergences in a properly diversified dual-execution environment, which does not require maintaining or checking execution metadata. We design a practical and deterministic dual-execution engine, a co-design for dual-execution and fuzzers, bug-sensitive diversification, comprehensive, and efficient divergence detection to ensure the effectiveness of Expozzer. The results of evaluations show that Expozzer can detect not only CVE-assigned vulnerabilities reliably, but also new vulnerabilities in well-tested real-world programs. Expozzer is 10 times faster than MemorySanitizer and is similar to AddressSanitizer.

Index Terms—N-version programming, dual-execution, fuzzing, sanitizers, bug detection

1 Introduction

Fuzzers test the target programs by mutating or generating inputs, and observing abnormal program behaviors. Fuzzing has become an essential means for software testing, and we have witnessed its impact. For example, OSS Fuzzer, a continuous fuzzing service built by Google, has already found over 14,000 bugs in 200 open source projects as of Aug, 2019 [1]. An effective fuzzing-based bug-finding approach requires two parts. (1) Path exploration. A fuzzer should explore paths to cover as much code as possible to reach potential bugs. (2) Bug exposing. Once a bug is reached, the approach should be able to faithfully expose the bug even if it does not exhibit observable behaviors (e.g., crashing the program).

Existing fuzzing research has primarily focused on improving path exploration and its performance, but not bug exposing. Path-exploration techniques include symbolic execution (e.g., Driller [2]), dynamic taint analysis (e.g., VUzzer [3]), concolic execution (e.g., QSYM [4]), heuristic-based static analysis (e.g., [5]), and many others [6], [7], [8], [9]. In addition, researchers have attempted to guide fuzzers to reach specific code locations. For example, AFLGo [10] implements guided grey-box fuzzing using static analysis to find pre-identified potential vulnerable points. SemFuzz [11] navigates fuzzing to the potential location of vulnerabilities extracted from the Git commit information.

By contrast, bug exposing is a less-studied topic, which is arguably because people believe that sanitizers such as ASAN [12] and MSAN [13] are effective in exposing bugs. As such, sanitizers have been extensively used and become the de facto standard for fuzzing. However, we identify a number of inherent problems with sanitizers that have impeded the power of existing fuzzers. First, sanitizers such as ASAN and MSAN are incompatible to each other, thus can not be applied to a target program together. This is a fundamental design problem—sanitizers employ a shadow memory–based scheme to maintain and check against metadata, which requires dedicated and exclusive uses of the same memory regions. As a result, fuzzers have to apply each sanitizer separately to detect specific classes of bugs, which inevitably wastes fuzzing resources. Second, sanitizers suffer from high performance overhead. Both ASAN and MSAN can have two to three times performance overhead [12], [13], which is considerably high in testing. Our evaluation in Section 6 shows that the performance overhead can be even worse when applying ASAN and MSAN on real world programs other than benchmark programs such as SPEC 2006. Third, bugs exposed by sanitizers can be non-exploitable. For example, over-written data in the date segment that is never used is not exploitable. Further, sanitizer-exposed bugs can be false positives because most sanitizers have stricter security enforcement comparing to specification of programming languages. All these limitations of sanitizers impede the power of coverage-based fuzzers.

To address the inherent limitations with existing sanitizers, in this paper, we propose Expozzer, a new bug

• *The authors are with the University of Minnesota, Minneapolis, MN 55455 USA. E-mail: {wang8330, kjlu, wu000273, pakki001}@umn.edu.*

Manuscript received 27 Aug. 2020; revised 6 May 2021; accepted 7 May 2021. Date of publication 13 May 2021; date of current version 2 Sept. 2022.
(Corresponding author: Aditya Pakki.)
Digital Object Identifier no. 10.1109/TDSC.2021.3079857

exposer that can comprehensively, efficiently, and faithfully report bugs when they are covered. The key intuition behind EXPOZZER is that a triggered bug will likely cause divergent behaviors (e.g., control-flow or data-flow) in a properly diversified execution environment. By comparing the behaviors, we are able to report the divergences and thus the bug. For example, a buffer over-read will obtain different data when the memory between buffers is randomized. With the intuition, EXPOZZER employs diversified dual-execution to report bugs, which does not require the maintaining and checking of metadata. To ensure the effectiveness and performance of EXPOZZER, we develop multiple techniques. (1) A practical and deterministic dual-execution engine. It captures non-determinisms and synchronizes two execution instances. (2) A co-design of the dual-execution and fuzzers that ensures fuzzing efficiency. (3) Bug-sensitive diversification. It diversifies memory and memory layouts of variants to effectively expose bugs. (4) Comprehensive and efficient divergence detection. It efficiently captures divergences in both control-flow and data-flow, and automatically locates the root causes of the divergences, i.e., the bug.

EXPOZZER has a number of advantages over sanitizers in bug exposing. First, EXPOZZER does not require shadow memory for maintaining and checking metadata; instead, it detects divergent behaviors in the end of execution, which avoids the compatibility issues and expensive tracking (thus ensuring the performance). Second, by enforcing various existing or new diversification schemes, EXPOZZER can detect a wide range of bugs including out-of-bound access, use-after-free, uninitialized uses, etc. The detection of all these classes of bugs can be realized in the same fuzzing round. Third, EXPOZZER's bug reports are faithful. Divergences are exhibited behaviors in control-flow or data-flow, so the detected bug are exploitable. Further, as long as non-determinisms are synchronized (see Section 6.3), EXPOZZER does not have false positives.

We have implemented EXPOZZER and integrated it with existing fuzzers to achieve automatic bug exposing. We also evaluated the effectiveness and performance of EXPOZZER. We collect 117 CVE-assigned bugs reported by sanitizers and apply EXPOZZER to detect them. We found 12 of them are non-exploitable bugs (e.g., overwritten data can never be used). For the remaining 105 bugs, EXPOZZER reliably detects 98 of them and can probabilistically detects 7 (depending on if divergences are triggered). Further, by applying EXPOZZER to well-tested programs, we also found 24 previously unknown bugs of various classes. As of the paper submission, 6 of them have been assigned with a CVE. Note that all these bugs are exploitable and can divert control flow and/or data flow. Finally, we compare the fuzzing throughput of EXPOZZER with ASAN and MSAN; results show that EXPOZZER significantly outperforms MSAN—ten times faster—and performs similarity as ASAN. The results confirm that EXPOZZER can comprehensively, efficiently, and faithfully detect exploitable bugs. We believe that, with EXPOZZER, we can unleash the power of existing fuzzers in finding bugs.

We make the following technical contributions:

- We identify inherent limitations with sanitizers in exposing bugs and propose to use diversified dual-execution to comprehensively, efficiently, and faithfully expose exploitable bugs. We also propose a co-design of dual-execution and fuzzers that ensures fuzzing efficiency.
- We develop multiple techniques, including practical and deterministic dual-execution engine, bug-sensitive diversification, comprehensive (both control-flow and data-flow) and efficient divergence detection, to ensure the effectiveness and performance of EXPOZZER.
- We implement EXPOZZER and integrate existing fuzzers into EXPOZZER to detect exploitable bugs. We found 24 new exploitable bugs of different classes in well-tested real-world programs, and obtained 6 CVEs for them.

The rest of this paper is organized as follows: Section 2 discusses the mechanism of sanitizers and their pitfalls, then presents the motivation of our work. Section 3 demonstrates the overall architecture of EXPOZZER and key techniques. Section 4 shows the design of EXPOZZER and how we combine the dual-execution system with existing fuzzers. Section 5 provides interesting details of implementation. Section 6 shows the results of evaluations. We briefly talk about possible ways to improve the performance of EXPOZZER in Section 7. Finally we discuss related works in Section 8 and conclude the paper in Section 9.

2 BACKGROUND AND MOTIVATION

Sanitizers are the de facto bug-exposers in fuzzing. There are four commonly used sanitizers: AddressSanitizer (ASAN), MemorySanitizer (MSAN), UndefinedBehaviorSanitizer (UBSAN), and ThreadSanitizer (TSAN). Each sanitizer is capable of exposing specific classes of bugs. ASAN detects stack overflow, heap overflow, use-after-free, global buffer overflow, etc. MSAN focuses on uninitialized-use bugs. UBSAN instead targets undefined behaviors such as misaligned pointers and signed integer overflow. TSAN captures data races in programs. ASAN and MSAN are the most widely used; they both employ a shadow memory-based method to maintain and check against metadata. Both sanitizers include a compiler instrumentation pass and a runtime library. We will discuss sanitizers' inherent problems mentioned in Section 1.

2.1 Pitfalls of Sanitizers

Although sanitizers are generally effective in bug exposing, they still have the following inherent problems.

Conflicts Between Sanitizers. The first problem of sanitizers is they are not compatible with each other because both ASAN and MSAN use the shadow memory-based detection mechanism. They allocate an overapproximated size of memory in the virtual memory layout at the beginning, thus leads to conflicts. Fuzzers have to apply sanitizers separately, which wastes fuzzing resources and may miss bugs within the given time budget. Moreover, sanitizers can be hard to use and deploy. For example, to use MSAN and to avoid false reports, all dependencies of the target program should be compiled with MSAN instrumentation, which is a non-trivial task in real-world testing [13].

Performance Overhead. Sanitizers are slow. Although both MSAN and ASAN claim to have moderate performance overhead—about 2x for ASAN [14] and 3x for MSAN [15], our test shows that, when fuzzing real programs, both ASAN and MSAN are much slower even when the target

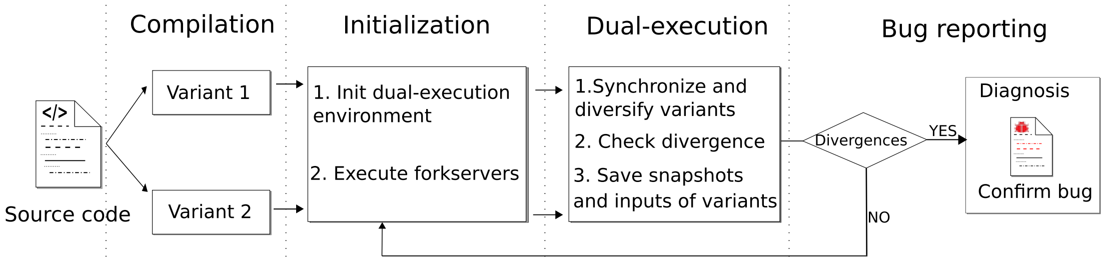


Fig. 1. Overview of EXPOZZER.

programs are small. For example, when using MSAN to test for uninitialized-use bugs, even for small programs without instrumenting any library, MSAN can slow down the fuzzing speed by more than 20 times. If testers apply each sanitizer to test programs sequentially, the inefficiency will dramatically slow down the bug-finding process.

Non-Exploitability. Sanitizers cannot report exploitability (i.e., exploited to cause security consequences) of detected bugs. Sanitizers may report unexploitable bugs because sanitizers detect any violation of specified rules such as “not using uninitialized memory”. According to Song *et al.* [16], the rules are stricter than the de facto programming language standard [16]. In some cases, the “violations” do not have any harmful impact. For example, over-reading limited bytes in the data segment that are never used does not cause security impacts. As we confirmed in Section 6.1, a number of studied sanitizer-reported bugs are non-exploitable. While detecting and fixing non-exploitable bugs typically do not hurt the security of the target program, they do incur maintenance burdens and may de-prioritize the fixing of critical exploitable bugs.

False Reports. Sanitizers can also generate false positives and false negatives. For example, out-of-bound accesses that skip the redzones will not trigger the detection of ASAN. Also, reading and copying uninitialized memory internally is common and typically harmless, so MSAN must manually handle some libc functions (e.g., `memcpy()`). Using predefined interceptor functions and hand-written assembly is also a source of false positives in MSAN [17]. If they are not handled correctly, MSAN may also generate false positives and false negatives because initialization status cannot be correctly maintained or checked.

2.2 Exploitability Reasoning

After sanitizers report bugs, we can also use automated reasoning techniques to test their exploitability. For example, Mayhem employs symbolic execution to determine if a bug can change control-flow transfers [18]. AEG [19], CRAX [20], and others [21], [22], [23] can also automatically determine the exploitability and even generate exploits. Although the tools mentioned above can reason about exploitability, they still have shortcomings. First, symbolic execution has its own limitations;they cannot scale to large real-world programs and easily lead to path explosion. Second, they primarily focus on bugs that can cause control-flow hijacking. But in real programs, bugs that are not related to control data, e.g., data-only attacks [24], [25], are becoming prevalent since control-data related bugs have been extensively studied [26]. It is not clear yet if these tools can also automatically test the exploitability for data-only attacks.

3 OVERVIEW

We propose EXPOZZER, a new bug exposer, to address the limitations with existing sanitizers. EXPOZZER is able to expose bugs comprehensively, efficiently, and faithfully when they are reached. We design and implement several techniques in a dual-execution system to make it practical and easy to use. We also enforce diversification in the dual-execution system and efficiently detect divergences introduced by bugs. In this section, we present the high level overview of EXPOZZER.

3.1 The Workflow of EXPOZZER

We first present the workflow of EXPOZZER, as shown in Fig. 1. There are four phases in EXPOZZER described as follows.

Compilation. In the compilation phase, the source code of the target program is compiled twice with different options (e.g., with and without enabling safe stack) for diversification and randomization schemes to generate diversified code variants. These options are not for compiler optimizations but for diversification. As will be shown in Table 2, we enforce four randomization schemes (e.g., SafeStack [27]) to diversify one variant but not the other variant. More details are shown in Section 4.3. Note that more diversification schemes can be adopted and enforced to both variants based on more needs.

Initialization. The initialization phase accomplishes two jobs: initializing the dual-execution environment and starting the execution of forkservers (see Section 4.2). For the initialization, EXPOZZER creates data structures for dual-execution, control-flow recording, and synchronization of the two variants. Then it starts the execution of variants.

Dual-Execution and Divergence Checking. EXPOZZER does multiple things during the execution of variants. First, it

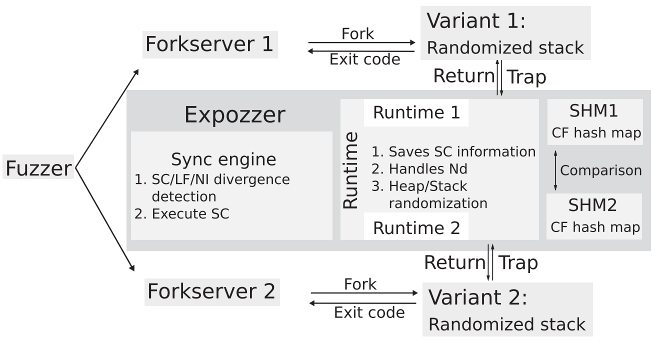


Fig. 2. Architecture and components of EXPOZZER. SC = System call, SHM = Shared memory, Nd = Non-determinisms. LF = Library functions, NI = Nondeterministic instructions.

synchronizes and diversifies variants, and records control- and data-flow information. Second, it performs divergence checking against the recorded control and data-flows. Third, if any divergences were detected, EXPOZZER saves snapshots of variants at the divergence point to enable further analysis and diagnosis.

Bug Reporting and Fuzzing Loop. At the end of each round of execution, upon divergences, EXPOZZER saves the current inputs and invokes sanitizers and debuggers to confirm the bugs. Also, EXPOZZER will clean up data structures for the current round and start new round of fuzzing.

3.2 A Co-Design of Dual-Execution and Fuzzer

Both the dual-execution and the fuzzers have their own specific designs. To realize EXPOZZER, we must propose a co-design of them that ensures fuzzing efficiency. The top two execution modes of fuzzing systems are dummy mode and forkserver mode. The dummy mode starts a brand new process for every round of fuzzing. Although dummy mode is easier to implement, it suffers from performance issues as it requires an initialization for every round of fuzzing. On the other hand, in forkserver mode, the fuzzer first invokes the target program as a forkserver. The forkserver finishes the initialization and stops at the beginning of the main function. When a new round of fuzzing starts, the forkserver spawns a new process, avoids duplication by performing an efficient copy-on-write mechanism between parent process and child process. We trade-off simplicity of design for fuzzing performance, by adopting the forkserver mode, in EXPOZZER.

Considering the efficiency of the forkserver mode, we propose to maintain two separate forkservers for spawning the variants; the architecture of the co-design is shown in Fig. 2. EXPOZZER coordinates both forkservers and variants and their communication (through shared memory). The control flow coverage information from the variants is compared and returned back to the fuzzer through shared memory. In addition to the coordination, EXPOZZER also diversifies, synchronizes, and monitors the execution of variants. More details are presented in Section 4.

With the co-design, some conflicts exist between the forkserver mode and the dual-execution system. Specifically, the return value (i.e., the `pid`) of the system call `fork()` is always different in the two variants, so it is one source of the non-determinisms which can introduce false positives. To handle the non-determinism, EXPOZZER intercepts `fork()` system call to return the same `pid` for both variants. However, the forkserver mode requires real `pid`s for both variants to control the execution. We thus instrument the calls of `fork ()` during comparison to distinguish them at runtime.

3.3 Key Techniques

There are many obstacles to overcome in order to make the dual-execution practical and easy to use, while comprehensively, efficiently and faithfully exposing bugs through control-flow and data-flow divergences. We propose several techniques to tackle these challenges.

Efficient Dual Forkserver. The process of program initialization is expensive; the performance can be further degraded in EXPOZZER because of the dual-execution system. We design and implement a dual forkserver in EXPOZZER to eliminate overhead caused by repetitive execution of variants, thus preserving high performance. With the dual forkserver, EXPOZZER can continue to execute variants when divergences are detected instead of restarting the testing.

Bug-Sensitive Randomization. We adopted two memory randomization techniques to detect both spatial and temporal memory bugs: memory poisoning and memory layout diversification. EXPOZZER poisons the memory in variants with different values after `malloc()` and `free()`, so that when any poisoned bytes are used in either system calls or determining control flow, EXPOZZER can capture divergences to detect temporal memory bugs such as use-after-free and uninitialized uses. On the other hand, the memory layout diversification technique in EXPOZZER diversifies memory layout in both heap and stack of variants to detect spatial memory bugs.

Comprehensive Detection of Bug-Triggered Divergences. To comprehensively detect bugs, EXPOZZER captures divergences in both data flow and control flow. To detect divergences in data-flow, EXPOZZER intercepts system calls and compares their arguments because critical bugs typically involve system calls. This way, we can avoid expensive runtime data-flow tracking. To detect control-flow divergences, EXPOZZER records and compares the control-flow information in hashmaps during the execution. Any divergences in control flow will be reflected in the hash values.

4 COMPONENTS DESIGN OF EXPOZZER

In this section, we present the design of key components in EXPOZZER and techniques mentioned in Section 3.1.

4.1 Practical and Easy-to-Use Dual-Execution

To ensure that the dual-execution is practical and does not introduce false positives, we need to synchronize the two variants properly and handle non-determinisms.

Challenges. There are several challenges in using the dual-execution for fuzzing. First, to eliminate false positives in fuzzing, benign non-determinsms such as random number generation in variants must be handled. Second, the performance overhead of dual-execution must be reduced so that more tests can be done within a limited fuzzing time budget. Third, dual-execution must be able to continue executing variants even if divergences are detected. Naively terminating programs after the first divergence not only hides potential bugs, but also confuses fuzzers about the coverage information. To overcome these challenges, we present the following design.

4.1.1 Synchronization

The dual-execution engine of EXPOZZER is divided into two parts: the application side and the system side. In the initialization phase of EXPOZZER, the application side is attached to each variant, and it is responsible for system call interception and coordination with the system side. The system side serves as the synchronization engine. On each system call, it is notified by both variants; it is also responsible for actually executing the system call and returning results of system calls back to variants.

EXPOZZER achieves synchronization between variants using system call interception. During the execution of variants, when a system call is encountered, variants trap into

TABLE 1
Sources of Non-Determinisms and Interception

Source	Intercept methods/tools
System calls	Syscall filter (`libseccomp`)
Library functions	Dynamic linker (`LD_PRELOAD`)
Non-deterministic instructions	Code instrumentation

the application side of the dual-execution system. EXPOZZER saves all the arguments of system calls and related execution information. After saving those information, the application side notifies the system side, i.e. the synchronization engine. On the server side, service threads are created to communicate with variants.

4.1.2 Handling Non-Determinism

Non-determinisms can cause benign divergences, leading to false positives. There are three sources of benign non-determinisms, as listed in Table 1: system calls, library function calls, and instructions that return random values. Our goal of handling non-determinisms is to eliminate false positives while preserving the normal execution of programs. Numerous solutions exist to handle benign non-determinisms. One solution can return predefined values for these function/system calls and instructions. This is easy to do and should work in most cases. The key problem with this solution is that no matter how the predefined value is set, it could potentially affect the program's execution. Further, the predefined value may be not meaningful to the program. This solution causes the program to behave abnormally, leading to false reports. Another possible solution is to instrument function/system calls and instructions during compilation. This also requires modification of the OS to ensure the solution can return the same value for variants. Moreover, this solution has no effect on program's execution since it can guarantee to return real and meaningful values to variants.

We choose the second solution to handle non-determinisms—we first identify a list of non-determinisms, intercept them (see Table 1), and make sure they return the same value at runtime. Specifically, interception is achieved through syscall filter (`libseccomp`), dynamic linker (`LD_PRELOAD`), and code instrumentation. When non-determinism is encountered at runtime, variants will trap into the application side of the dual-execution engine and notify the server side, and the server side is responsible for actually executing system calls/functions or instructions, and returning the same value back to variants.

Supporting Multi-Threading Programs. Thread interleaving may also cause benign divergences in system-call synchronization, so we need to synchronize all shared resource/memory accesses. A number of works have attempted to achieve such synchronization. As shown in the deterministic multi-threading (DMT) [28], [29], [30], [31], this is achievable, but would significantly slow down the runtime. Considering that EXPOZZER's goal is to find bugs instead of defending against attacks, we choose to adopt the lightweight synchronization mechanism (mapping threads in the two instances) borrowed from Varan [32] and the system-call synchronization part of MVEE [29], [31]. Although, the synchronization is lightweight, we believe that it suffices for fuzzing—as shown in Section 6, EXPOZZER can support multi-threading programs, and we did not observe false positives in the testing.

4.1.3 Continuous Fuzzing

One important feature of EXPOZZER requires, not terminating the current execution, upon encountering a divergence. To support that, when a divergence is detected, we instruct EXPOZZER to save the snapshots of variants and related execution information on the divergence, and to continue the execution by discarding the corresponding seed input triggering the divergences. We discard such seed inputs for two reasons. First, it is to avoid redundant divergences because using the same seed input would repeatedly trigger divergences. Second, having a divergence means the program is already in an erroneous state. That is, we have already found the bug triggering the divergence. Even if new divergences are discovered by keeping the seed input, they are likely false positives caused by the same bug.

4.2 Efficient Dual Forkserver

A key design goal of applying the dual-execution in fuzzing is to preserve the performance. In order to achieve the design goal, unnecessary performance overhead must be eliminated. One of the most commonly used techniques in popular fuzzing systems is forkserver. The forkserver can save time of loading libraries at the initialization phase of the program. In fuzzing systems that employ forkserver, the target program is first invoked by fuzzer as a forkserver; after finishing the initialization, it stops at the beginning of the main() function. To start a new round of execution, the forkserver just forks itself and continues the execution to save the time of the initialization.

Naively applying the forkserver technique into the dual-execution system, however, would not work because of various reasons. First, the `fork()` system call introduces non-determinism into execution, thus can cause false positives. Second, simply intercepting `fork()` system call to make it return the same `pid` of child processes for both variants can make the management of dual-execution system harder since `pid`s are essential to managing different processes in the dual-execution system. For example, when variants timeout during fuzzing, `pid`s are necessary information to kill variants and start a new round of fuzzing. The reason for this conflict is enforcing the same interception policy for both the forkserver and the normal process of the target program. There are two places where fork can happen in EXPOZZER: forkserver (to spawn variants) and normal execution. They have different requirements on the return value of `fork()`. In forkserver, real `pid`s of `fork()` must be returned so that EXPOZZER can perform management. While in normal execution, it returns the same `pid` for `fork()` in variants to eliminate non-determinism. Therefore, the key to address the conflict is to distinguish `fork()` between the forkserver and normal execution and to return proper `pid` correspondingly.

To distinguish the sources of `fork()`, we build the dual forkserver in EXPOZZER by incorporating instrumentation and the runtime environment. At the compilation phase, when the customized compiler adds forkserver instrumentation to

TABLE 2
Randomization Techniques Enforced in EXPOZZER

Technique	Region	Error type
Memory poisoning	Heap	Temporal
Address randomization	Heap	Spatial & temporal
Random padding	Stack & heap	Spatial & temporal
SafeStack	Stack	Spatial

the target code, the compiler also adds an extra parameter to `fork()` used by the forkserver. The dual-execution engine will then intercept `fork()` during execution. In the intercepted version of `fork()`, EXPOZZER applies different interception policy if the extra parameter is presented. This way, in normal execution, EXPOZZER can return the same `pid`, and in the forkserver, EXPOZZER can return the real (different) `pid`s to variants.

4.3 Bug-Sensitive Randomization

Because EXPOZZER exposes bugs through divergences, it is essential for EXPOZZER to expose divergences as long as a bug is triggered. Since EXPOZZER targets memory errors (both temporal and spatial), we focus on memory and memory layout randomization. It is important to note that EXPOZZER is compatible with other randomization techniques (e.g., injecting random delays in thread scheduling). More randomization schemes can be plugged in to cover more classes of bugs. Our randomization schemes target both the stack and the heap, as summarized in Table 2. These randomization schemes are dedicated for exposing bugs, which are both efficient and effective in generating divergences for memory errors. Except the SafeStack, other three schemes are proposed in this work.

Heap Randomization. Our heap randomization includes address randomization, random padding, and memory poisoning. The heap allocator in EXPOZZER adds random padding bytes after a successful memory allocation on heap. The padding bytes are different in variants so that if values are used, divergences can be detected in data-flow and/or control-flow. The memory poisoning technique in EXPOZZER is used to assign poisoned values to memory regions that have just been allocated or freed. In this way, uninitialized-use and use-after-free bugs can cause divergences and thus be detected. To further increase the ability of detecting temporal memory bugs, EXPOZZER allocates heap object at randomized addresses on the heap instead of consecutive addresses. Because of the randomized allocation, poisoned areas would be less likely reused by other allocations.

Stack Randomization. The stack randomization in EXPOZZER gives variants different stack layouts. EXPOZZER employs a different randomization mechanism on stack comparing to heap. There are several reasons for not adopting the same randomization mechanisms as for the heap. First, a complete randomized allocation stack could degrade the performance of the dual-execution system. Unlike heap, accesses on stack are based on offsets and the base pointer, a complete randomized allocation scheme adds too much overhead to the system. Second, memory poisoning on stack may not be as effective as on heap. Stack is a small region of memory that are reused frequently; this is inherently different compared to heap as heap is a huge chunk of memory. Poisoned values on stack can be overwritten by later execution very quickly, thus rendering the technique ineffective.

To overcome these problems, EXPOZZER first adopts the SafeStack technique from CPI [27] to detect spatial memory errors. SafeStack divides the stack layout of a variant into two regions: safe and unsafe. The safe region stores variables that can only be accessed in a secure way, while the unsafe region stores buffers that can cause problems. Specifically, EXPOZZER employs clang's SafeStack instrumentation to realize the stack randomization. In EXPOZZER's compilation phase, one variant is compiled with SafeStack, and the other is compiled without it.

To further detect temporal memory errors on the stack, we also inject random padding between stack frames. Specifically, a random number of words (8 bytes each) will be injected between stack frames. To make sure that the execution can restore the stack upon a function return, the previous stack-frame pointer will be saved in the current stack frame, which also ensures that the two variants have different pointers on the stack. This way, the overlaps between the current and the previous stack frames will be randomized, enabling the detection of temporal memory errors.

4.4 Comprehensive Divergence Coverage

To comprehensively cover different classes of bugs, EXPOZZER detects divergences in both control-flow or data-flow. The control-flow divergence detection is done by combining compiler instrumentation and runtime checking. The data-flow divergence detection is realized through argument comparison of system calls. In this section, we present the details.

4.4.1 Control-Flow Divergence

There are mainly two methods to achieve control-flow divergence detection: online and offline. The online control-flow comparison requires the dual-execution system to synchronize variants at each conditional jump. An advantage of online control-flow comparison is that it can detect control-flow divergence on time, i.e. when the divergence happens. However, this property is only necessary when used in defense systems as it can immediately stop execution when abnormal behaviors are detected. In bug detection, it is acceptable for the target program to continue execution even if abnormal behaviors are detected. Another advantage of online control-flow comparison is that it can accurately report the location of control-flow divergences. However, this can be easily replaced by post-execution analysis based on debuggers or sanitizers.

Online control-flow comparison however comes with unacceptable drawbacks. First, the online detection can introduce unacceptable performance overhead. The performance overhead can degrade the throughput of fuzzing. Second, it can complicate the design of the dual-execution system. To realize the online control flow comparison, the dual-execution must enforce instruction-level synchronization between variants. This not only increases the performance overhead, but also introduces design challenges and problems. For example, to perform the comparison, the destination of control-flow transfer must be identified and mapped accurately between variants. This task can be hard to achieve since variants have different memory layouts because of randomization.

Offline Comparison of Control-Flow Hashes. Considering the drawbacks of online comparison, Expozzer instead employs a offline control-flow detection mechanism. To realize the offline comparison, control-flow information is recorded during execution. Expozzer first employs AFL's control-flow recording mechanism. During compilation, each basic block is assigned with a random block ID, and during execution, programs write control-flow information into the control-flow hashing map. Block IDs are used to determine which byte in the hashing map is updated when the corresponding control-flow transfer happens.

Synchronized Dual-Compilation Mode. There is however an issue in applying AFL's compiler instrumentation—since the randomized block IDs are generated during compilation, and Expozzer requires compiling the source code twice to have two variants with different stack layouts, the block IDs are different for the same block in two variants, which leads to false positives. To solve this issue, we introduce dual-compilation mode for compiling variants. In the dual-compilation model, the compilation processes are also synchronized, so that same block in variants will always have the same ID to eliminate false positives.

4.4.2 Data-Flow Divergence

Control-flow divergence detection by itself is not sufficient to expose various classes of bugs. Occasionally, bugs do not change control-flow at all but still have severe effects on programs such as information leak. Therefore, supporting data-flow divergence detection can detect these kinds of bugs. However, supporting data-flow divergence detection is non-trivial. The most challenging problem is the performance overhead of data-flow tracking. Traditional data-flow tracking methods combine static analysis and runtime tracking.

Static analysis is performed before execution to identify legitimate sources of data-flow for a pointer or variable. Runtime tracking and checking record data-flow information, can identify if the current source is legal. There are inherent problems with this scheme. First, static analysis cannot guarantee the soundness or completeness of identifying all the legitimate sources of data-flow. Second, to enforce this scheme, the tracking and checking need to be done at the instruction level. That is, we need to intercept memory read and write instructions and invoke the cross-variant comparison for each instruction, which will certainly introduce unacceptable runtime performance overhead.

Expozzer therefore employs system-call level checking to detect divergent data flow. Although system-call level divergence checking is not as fine-grained as instruction level, we believe it is adequate to detect critical bugs resulting data-flow divergences. First, the rationale behind this design choice is that critical operations are typically performed through system calls on modern operating systems, thus critical bugs are expected to trigger divergences in arguments of system calls. Bugs that do not involve system calls are hard to exploit to cause critical impacts. Second, based on our evaluation of 105 CVE-assigned vulnerabilities, Expozzer did not introduce any false negative caused by the system-call level (instead of instruction level) divergence checking. To the best of our knowledge, system-call level checking not only reduces the performance overhead (Table 4 in [33]) of divergence checking for data-flow, but also preserves the effectiveness of detecting exploitable bugs for most real-wold programs.

TABLE 3
Percentage of Memory Related Instructions and System Calls in a Random Sample of Real World Programs

Program	%read	%write	%Sys calls	Total instructions
base64	24.2	8.7	0.0003	305024
ffmpeg	29.1	21.2	0.00002	177062602
md5sum	23.3	8.1	0.0002	266824
objdump	24.1	19.7	0.0001	189940436
readelf	22.8	13.2	0.0003	11858122

To estimate the performance overhead, we measure the average number of memory-related instructions and system calls. Table 3 shows the percentage of memory-related instructions and system calls in our random sample of five real-world programs. We use Intel SDE and run these programs three times with different inputs. The empirical numbers suggest that implementing an instruction-level synchronization can introduce a high performance overhead in testing for real-world programs.

4.5 Bug Confirmation

Once a divergence is detected, Expozzer continues to confirm the actual bugs. In the testing phase, when divergences are detected, the corresponding inputs are saved. Expozzer thus employs existing tools for bug confirmation. Expozzer divides bug confirmation phase into two steps: the first step is Expozzer feeds input saved during the fuzzing phase to the target program running under Valgrind and check for error output; then Expozzer feeds the same input to the target program compiled with AddressSanitizer. Valgrind [34] can detect memory bugs in heap with very low false positive and false negative rates. Due to its high performance overhead, it is not feasible to use Valgrind in the real-time fuzzing phase. We use Valgrind as the confirmation tool to avoid its high performance overhead but benefit from its accuracy offline. We use Valgrind to detect heap related bugs and uninitialized uses on the stack. We employ AddressSanitizer to complement Valgrind for bugs involving stack overflow and bugs on the heap.

5 Implementation

We present interesting implementation details of Expozzer in this section. We first introduce the implementation details of integrating the dual-execution system with fuzzers, then cover some interesting details such as system-call divergence detection, non-determinism handling, dual fork-server, and dual compilation.

5.1 The Dual-Execution Engine

There are four components in the dual-execution system: monitor, interceptor, synchronization engine, and divergence checker.

The Monitor. The monitor is responsible for managing the dual-execution system. After the initial stage of Expozzer, the fuzzer forks the monitor instead of the forkserver of the target program. The monitor sets up execution environment for the dual-execution system. It first creates shared memory regions for recording control-flow information for variants. Then it creates and initializes shared memory regions for saving dual-execution related information for variants. After setting up necessary shared memory regions, the monitor pre-loads the interceptor, forks forkservers and the synchronization engine for them. Except for initialization, the monitor also sets up communication to variants and the synchronization engine.

The Interceptor. The interceptor is in charge of intercepting system calls, and communicating between variants and the synchronization engine. During the execution of variants, when system calls are encountered, variants trap into the interceptor. The interceptor copies system call arguments and related execution information into the shared memory region between the variant and the synchronization engine. After saving the information, the interceptor notifies the synchronization engine that a system call in the corresponding variant has already been invoked and waits for the engine. The interceptor forwards the return value of system calls from the synchronization engine to the variant so that the execution can be continued.

The Synchronization Engine. The synchronization engine synchronizes variants on system calls and performs divergence detection. In the engine, one thread is responsible for communicating with one variant. Upon system call, the interceptor saves information and notifies the engine. When the engine receives the notification, it fetches system call information from shared memory regions and performs divergence detection.

The Divergence Checker for System Calls. There are two ways to compare corresponding arguments from variants: memory check (i.e., non-pointer type arguments) and address check (i.e., pointer type arguments). To pass the address check, two arguments must be both non-null or null. Because two variants have different memory layouts, blindly comparing pointer values from different variants leads to false positives. On the other hand, the memory check compares the concrete value pointed to by the arguments. We design a flexible divergence detection component inside the engine to avoid hard-coding comparison policies into source code. At the initialization phase of the engine, a configuration file containing comparison policies is read. After the divergence detection in system call arguments, if there is no divergences in arguments, the engine executes the system call requested by variants and passes the return value back to variants. If any divergences are detected in arguments, the engine notifies the monitor. The monitor saves snapshots of variants and related execution information in shared memory regions. After saving these information, the engine proceeds in a way similar to the case when no divergence is detected. Note that the divergence checking for control-flow hashes is at the end of the current execution.

5.2 Identifying and Handling Non-Determinisms

As mentioned in Section 4, there are three sources of non-determinisms: system calls, library function calls, and instructions. We have a conservative strategy for finding the non-determinisms. For library function calls, we summarize potential sources of non-determinisms in library function calls into two categories: time/date, e.g., `gettimeofday()`, `clock_gettime()`, `time()`; process information, e.g., `getpid()`. Similar to library functions, we identify two kinds of nondeterministic instructions: time, e.g., `rdtsc`; random number, e.g., `rdrand`. For system calls, except the ones related to memory layout such as `mmap`, all other systems calls are selected for synchronization.

Once we identify the non-determinism sources, we intercept them and force them to return the same values to the two variants. Our interception employs different approaches to intercept system calls, library calls, and instructions. For system calls, we choose to temporarily patch the syscall table using a kernel module. This way, extra context switches can be avoided to improve performance. For library calls, we choose to patch the GOTPLT table that contains the entries to these library calls. For non-deterministic instructions such as `rdtsc` and `rdrand`, we replace them with an one-byte interrupt instruction (e.g., INT3) which allows us to perform the normalization by handling the interrupt.

5.3 Dual Compilation

Since EXPOZZER employs control-flow hashing for control-flow divergence detection, the random number generation process in control-flow hashing must be taken care of. In AFL instrumentation, each basic block is assigned an ID so that the write location in the hashmap can be determined by IDs of the current basic block and previous basic blocks. The block ID assignment is done through random number generation during compilation: for a source code file, a random seed is first generated based on (1) the `pid` of the compiler and (2) the current time. Then the seed is used to generate a series of random number used as block IDs. To avoid false positives in control-flow divergence detection, we must make sure that the same basic blocks in different variants must be assigned the same block ID. We solve this problem by synchronizing the random seed generation in compilation in EXPOZZER, so that false positives in control flow divergence detection are avoided.

5.4 Integrating Fuzzers

In EXPOZZER, we combine three different fuzzers with EXPOZZER to find bugs: AFL, FairFuzz, and AFLPP. Although they are different fuzzers, they all use basic AFL execution model. When fuzzing with AFL, the forkserver of the target program is forked by AFL. After being forked, the forkserver performs all the initialization work, stops at the beginning of `main()` function, and waits for the commands from AFL. To start a new round of testing, the forkserver spawns itself and starts running. EXPOZZER follows the same execution model when integrating dual-execution and other fuzzers. Note that for performance comparison with sanitizers, we specifically use AFLPP, and the details are presented in Section 6.4.

6 EVALUATION

We evaluate EXPOZZER in the following perspectives.

1) False negatives. We evaluate if Expozzer may miss valid bugs due to its divergence-based detection.
2) Effectiveness. We evaluate if Expozzer can detect known and previously unknown bugs.
3) False positives. We evaluate if bugs detected by Expozzer can be false positives.
4) Performance. Performance is important for fuzzers to test more inputs within a given time budget. We compare the performance of Expozzer with sanitizers.

Experiments are conducted on a machine equipped with two Intel Xeon Platinum 8268 CPUs; each CPU has 24 physical cores. The machine has 256 GB physical memory and Ubuntu 18.04 LTS as the installed OS. To evaluate Expozzer on real programs, we test each program within a virtual machine. The virtual machine has 4 GB memory and 2 CPU cores.

6.1 False Negatives

To evaluate the false negatives of Expozzer, we perform two sets of experiments. We test if Expozzer can detect both CVE-assigned vulnerabilities and bugs injected by LAVA [35].

Testing With CVE-Assigned Bugs. To test the false negative rate of Expozzer, we first collected 117 CVE-assigned vulnerabilities that were detected using sanitizers. Reproducing previously reported CVEs is non-trivial. To speed up the process and increase the chance of reproducing CVEs, we extract all the CVEs related to binutils. The programs represent the most common categories encountered in real world: terminal programs, encoding/decoding programs, and encryption/decryption programs. We picked binutils, as the code is self-contained, and its dependencies include the basic libc libraries. This increases the probability of reproducing the vulnerabilities. More importantly, they are extensively used in other fuzzing research papers; focusing on such programs would allow us to test if Expozzer can find bugs in even well-tested programs and compare it with other fuzzers.

As of the paper submission, there are a total of 180 CVEs related to binutils, according to the CVE website [36]. In addition to the binutils-related CVEs, we also obtained the list of CVEs from [37], [38]. After exhausting the CVE reports from the two lists, we finally manually reproduced 117 CVEs covering all major categories of memory bugs including: global/stack/heap buffer overflow, use-after-free, null pointer dereference, etc. The number of real programs used in evaluation is reasonably comparable to other fuzzing works. The reproducing work is actually laborious. There are many reasons for not being able to reproduce, such as requiring special inputs, a specific system configuration, a specific thread interleavings, etc. It is worth noting that the reproduction is manual, which is independent of the Expozzer or any other fuzzers.

In the experiment, we found that 12 of the CVEs are non-exploitable. These non-exploitable bugs out-of-bound read or write buffers with a very small number of bytes. We carefully reviewed the source code involved by these bugs. One can confirm the bugs to be non-exploitable as the over-written part does not target critical data and is never used, or the over-read data is never used.

For the remaining 105 cases, we found that Expozzer can reliably detect 98 of them and detect 7 (6.7 percent) of them at random. We looked into the probabilistic cases and found that they are bugs that write to the heap out-of-bound. In these bugs, the write does not change any important variable or data structures on heap, thus failing to cause a divergence during execution. A divergence is triggered if other heap objects are allocated close to the overflow object. Note that we can further reduce the number of probabilistic cases by enforcing other diversification schemes, as we will discuss in Section 7.2.

Testing With LAVA. We further evaluated the false negatives of Expozzer with the LAVA-1 dataset [35]. The LAVA-1 dataset contains 69 bugs inserted into the file program in coreutils. We successfully reproduced 68 out of 69 synthetic bugs. We tested these 68 bugs with the given PoC in LAVA-1, and the results show that Expozzer can capture all of the 68 bugs. Based on this experiment, Expozzer does not have any false negative in normal crash-based bug detection. The bug we failed to reproduce was not triggered with given PoC when compiled using clang with vanilla AFL's instrumentation(without Expozzer's instrumentation) on various versions of Ubuntu OS. We believe that the failure is not caused by Expozzer's instrumentation.

6.2 Detecting New Bugs on Real Programs

To validate the effectiveness of Expozzer, we run it on well tested programs. We run Expozzer in virtual machines; each virtual machine is set to have 2 cores and 4 GB memory, we run the experiments in only 24 hours and then stop them. The tested programs are shown in Table 5. In total, 24 new bugs are found during the test, as shown in Table 4. Categories of bugs detected by Expozzer include uninitialized use, heap overflow, invalid memory access, and null pointer dereference. We are confident to conclude that Expozzer is able to find different categories of bugs in real-world well tested programs.

6.3 False Positives

As shown in Sections 6.1 and 6.2, we have applied Expozzer to detect known and new bugs in widely tested programs. During the course of the evaluation, we did not observe any false positives, which confirms that, by handling the non-determinisms, Expozzer can faithfully and precisely expose bugs. As shown in Table 5, some tested programs are multi-threaded; during the evaluation, Expozzer does not report any false positives as well and successfully detects a bug in `ffmpeg`.

6.4 Performance Comparison With ASAN and MSAN

We run Expozzer, ASAN, and MSAN with same compilation parameters on same target programs, and monitor the fuzzing throughput of these three detection methods in executions per second. The reason we compare Expozzer to ASAN and MSAN is that they are the de-facto bug "exposers" used together with fuzzers. It is worth noting that Expozzer is for exposing bugs instead of for exploring program states. Bug exposing and program state exploration are two paralleled research fields in fuzzing. We can choose any possible combination of exposing and exploration methods, which only incurs engineering efforts.

For Expozzer, we use the dual-execution environment. However, for ASAN and MSAN, we use the single-execution environment, e.g., the traditional sanitizer-based fuzzing

TABLE 4
New Bugs Found by EXPOZZER in Real Programs

Program	Total	UU	HO	NPD	SO
mysofa2json	7	1	5	1	0
ffmpeg	1	1	0	0	0
readelf	1	1	0	0	0
bento4	13	8	3	1	1
cjpeg	2	2	0	0	0

UU = Uninitialized use, HO = Heap overflow, NPD = Null-pointer dereference, SO = Stack overflow.

settings. The fuzzer we use for throughput testing is AFLPP, the results are shown in Table 6. It is clear from the table that the throughput of EXPOZZER is close to ASAN. However, EXPOZZER is at least 10 times faster than MSAN in most target programs. Considering the fact that in real-world testing, to cover common memory bugs, testers must combine ASAN and MSAN together, EXPOZZER would significantly outperform the combination of ASAN and MSAN.

The only target that ASAN outperforms EXPOZZER significantly is `readelf`. The reason for the bad performance in `readelf` is that `readelf` issues too many system calls (typically more than 2,000) when the fuzzer mutates the inputs. The invalid inputs cause `readelf` to constantly find marker bytes using the system call lseek(). Because system call synchronization is the most time consuming operation in EXPOZZER, the frequent system calls lead to the bad performance in the case of `readelf`.

7 DISCUSSION

Although EXPOZZER is effective in finding exploitable bugs, we discuss the strengths and limitations of our current implementation in this section. Moreover, we also propose solutions for the identified limitations for potential future work directions.

TABLE 5
Command Line Parameters to Invoke Target Programs

Program	Command Line
readelf	readelf -a @@
objdump	objdump -D @@
size	size -t @@
nm-new	nm-new -A -D @@
objcopy	objcopy @@ test
speexenc	speexenc @@ -
ffmpeg*	ffmpeg -hide_banner -loglevel quiet -i @@ -f null /dev/null
avplay*	avplay -loglevel quiet @@
mysofa2json	mysofa2json @@
djpeg	djpeg @@
cjpeg	cjpeg @@
mp4info	mp4info –verbose –show-layout –show-samples –show-sample-data @@
mp4decrypt	mp4decrypt @@ /dev/null
mp4dump	mp4dump –verbosity 3 @@
avcinfo	avcinfo –verbose @@

*: *Multi-threading programs.*

TABLE 6
Throughput Comparison, in Terms of Executions per Second (e/s), Across Vanilla AFL, ASAN, MSAN, and EXPOZZER

Target	Vanilla AFL	ASAN	MSAN	EXPOZZER
	(e/s)	(e/s)	(e/s)	(e/s)
readelf	381	137	34	32
size	800	420	37	520
objcopy	1000	350	34	280
objdump	70	35	18	35
nm	900	400	36	460
djpeg	1100	500	37	400
cjpeg	3000	840	37	720
pngfix	1600	730	37	600
tiffdump	3150	1350	38	1020
Average (%)	100%	42.2%	6.1%	37.0%

Except that EXPOZZER uses the dual-execution environment, the other cases use a single-execution environment.

7.1 Uninitialized Use Detection

We noticed that uninitialized-use bugs are prevalent among bugs found by EXPOZZER. After carefully analyzing and comparing EXPOZZER with existing techniques on detecting UUM bugs, we think EXPOZZER has advantages over other approaches especially comparing to MSAN and Valgrind. First, MSAN is hard to use in real testing [13] because in order to eliminate false positives, testers need to compile all the dependencies of the target program, which may not be a trivial task especially when the target program has complicated dependencies. This can prevent testers from using MSAN in real testing. Second, the performance of MSAN and Valgrind are not satisfying. In Section 6.4, we have shown that MSAN introduces more than 10 times slow down for real programs even if the programs are simple. Valgrind has worse performance as it usually introduces a slow down between 10x–50x, and it could be even worse when testing multi-threaded programs.

Another advantage of EXPOZZER compared to MSAN is, the divergence based bug detection methodology can be deployed without requiring the source code. The source code of the test program is required to deploy MSAN in real testing, not to mention that testers also need to prepare the source code of any dependencies the target program has. The source code preparation may be hard or even impossible when testing commercial software. EXPOZZER's divergence-based methodology can be used without source code. The tool already utilizes heap randomization without source code, and could potentially achieve stack randomization without source code by modifying QEMU and enabling AFL's binary fuzzing mode.

7.2 Reducing Probabilistic and False-Negative Cases

In EXPOZZER, memory layouts are diversified to cause divergences when there is a memory bug during the execution. Although effective in its current form, it is possible to further diversify memory layouts to increase the sensitivity of divergences. EXPOZZER's current randomization technique does not perform well in heap/global out-of-bound write. Allocations of heap objects are dynamic, if the out-of-bound write on the heap does not overwrite any important data structures, no divergences will be introduced, leading to false negatives. To improve the detection ability of EXPOZZER, especially on out-of-bound write on heap, more randomization techniques could be used. For example, to detect heap out-of-bound write, EXPOZZER could introduces more randomization on heap, e.g., allocating memory regions close to each other while they have different orders on heap, so that when out-of-bound write happens, it will overwrite different data on heap to cause divergence. Similar techniques could also be used in global data segment.

7.3 More Diversification Schemes

Another possibility for EXPOZZER is to introduce more randomization other than memory layouts. For example, scheduler can be randomized to detect concurrency bugs in multi-threaded programs. After the scheduler is randomized, EXPOZZER may need to change synchronization mechanism: enforcing strict system call synchronization and comparison may cause a large number of false positives and introduce problems for dual-execution. To eliminate false positives introduced by schedule randomization, EXPOZZER may need to loosen the system call synchronization and perform comparison until the execution is finished.

A more aggressive diversification technique involves mutating the program to detect semantic bugs. EXPOZZER could first run static analysis on the target program and identify critical parts, e.g., parts with security checks or configurations. Then EXPOZZER changes one variant by deleting or adding one critical part. After the mutation, EXPOZZER could perform fuzzing on the target program and check if any divergence is detected. If a divergence is detected between the normal variant and the mutated variant, it could indicate a semantic bug, e.g., missing security check or mis-configurations.

7.4 Integrating More Fuzzers

Although the evaluation shows that EXPOZZER is very effective on detect memory-related bugs through divergence occurs in control-flow and data-flow, its detection ability is limited by the fuzzers integrated in. During the experiment, we observed that the coverage of our existing fuzzers is not good; this causes EXPOZZER unable to explore many interesting paths in the target program. This can be solved by integrating a better fuzzer into EXPOZZER. To integrate a fuzzer with advanced techniques such as concolic execution, the dual-execution model may need to be changed accordingly.

7.5 Supporting More Types of Bugs

The security community tends to define vulnerabilities based on the impacts of semantic errors (or bugs). For example, a missing bound-check bug is about a semantic error (i.e., the root cause), and a buffer overflow can be the impact of a missing bound-chek bug. Memory errors are a broad class of security-critical vulnerabilities, which are defined based on the impacts instead of the root semantic errors. EXPOZZER has been able to expose the broad class of memory errors, including buffer overflows, use-after-free, and uninitialized uses, regardless of the root semantics errors. That is, no matter what types of the bugs are, as long as they lead to the memory errors, EXPOZZER can detect them. In fact, the core idea of EXPOZZER is to expose bugs based on their unexpected behaviors or incorrect results. This generally holds for most types of bugs because by their nature, bugs tend to result in unexpected or incorrect results. Therefore, EXPOZZER can be further extended to expose more types of non-memory bugs by checking the corresponding behaviors and results.

7.6 Use EXPOZZER as Fuzzing Feedback

A fuzzing framework consists of two orthogonal parts: exploration and bug exposing. EXPOZZER focuses on the exposing part. Existing fuzzers mainly use code coverage as a feedback to guide the exploration. Indeed EXPOZZER offers extra dimensions other than code coverage, such as internal data flows and outputting results, that can indicate if the fuzzer is making progress. We would like to keep the use of EXPOZZER for fuzzing feedback for future work.

7.7 Further Exploitability Assessment

Developers can further investigate the exploitability of a bug by looking into the report generated by EXPOZZER. EXPOZZER's report tells developers how many divergences are detected during the execution in system calls and if there is control- or data-flow divergences. Developers can know what system calls are involved in divergences by manually look into core-dumps generated by EXPOZZER and the error reports generated by our customized debugger. Further severity and exploitability of detected bugs can be determined by the investigation. For example, in the investigation, if developers found out that the heap out-of-bound read or uninitialized bytes are used in system calls such as being used as the buffer in write(), it can indicate a potential information leakage on heap, or if the bug-related bytes are used to determine the control-flow, or used as the length parameter in write(), it could cause an arbitrary memory read.

8 RELATED WORK

In this section, we compare EXPOZZER to research works involving multi-variant execution, software diversification, and bug detection.

Multi-Variant Execution System. Cox *et al.* [39] proposed N-Variant system and define multi-execution system as a combination of a polygrapher and a monitor. The polygrapher diversifies variants while the monitor performs the checking during the execution. Basile *et al.* [40] implemented replicated execution of multi-threaded application by using a loose synchronization algorithm, eliminating non-determinism, while still preserving the concurrency of the original program. ReMon [31] improves the performance of multi-variant execution system by adopting cross-checking method for security critical system calls and releases the monitoring for other system calls. GHUMVEE [41] can handle real world multi-threaded programs by spawning for every set of threads during multi-variant execution. It intercepts the user-space synchronization operations and solves the synchronization of threads, thereby avoiding deadlocks.

Software Diversification. Cohen *et al.* [42] employed 14 diversification techniques such as replacing instructions with equivalent ones, instruction reordering, variable substitution, adding and removing jumps to make the protected software more secure. Forrest *et al.* [43] argued that there are many unnecessary consistencies existing in modern computer systems, thus proposed dead-code elimination/insertion, code reordering, stack frame padding/location randomization, runtime-check randomization etc. to protect software. They also disrupt buffer overflow attacks by adopting stack frame allocation randomization.

Pappas *et al.* [44] applied various code randomization techniques such as atomic instruction substitution, instruction reordering, register reassignment to protect software from ROP attacks. Larsen *et al.* [45] systematically studied existing diversification solutions, and generalized properties of diversification techniques into: what to diversify, when to diversify and the impact of diversity. These research works are orthogonal to EXPOZZER. It is always possible to implement more diversification techniques into EXPOZZER; the result presented in Section 6 has already shown that effective memory diversification can expose many bugs. We argue that further diversification can enhance EXPOZZER's ability to expose more bugs.

Differential Testing. One of the most recent published research work on differential testing is TimePlayer, a differential testing system working on binaries that can detect uninitialized variables in Windows by leveraging record & replay, differential testing and symbolic taint analysis [46]. EXPOZZER detects extra kinds of memory bugs except uninitialized use of memory by providing more kinds of randomizations and employs simpler confirmation techniques since EXPOZZER is a source-code based testing system. There are some other differential testing systems except TimePlayer, such as [47], [48], [49], [50], [51], [52]. All of the research works target specific applications e.g., compilers and crypto implementations. EXPOZZER is different from them as it is targeting memory corruptions bugs.

Exploit Generation. Mayhem [18] employs hybrid symbolic execution to automatically detect exploitable bugs. Hybrid symbolic execution leverages both online and offline execution so that memory is not exhausted. Mayhem uses index-based memory model to better reason about the memory status. CRAX [20] performs concolic execution on software following the failure directed path. Several techniques are integrated into CRAX to improve accuracy and speed such as whole system environment emulation. FUZE [23] determines exploitability of use-after-free vulnerabilities in kernel by combining fuzzing and symbolic execution.

Revery [22] leverages the layout-contributor digraph and fuzzing to search for exploits. SLAKE [21] employs both static and dynamic program analysis techniques to generate exploitation for kernel vulnerabilities. All of the research works listed above rely on heavyweight program analysis techniques using a combination of static, dynamic analysis, and symbolic execution, which may introduce unacceptable performance overhead or introduce potential false positives. Different from these works, EXPOZZER can report the exploitability of a given bug by capturing the divergence in control-flow and data-flow between diversified variants, with a low performance overhead.

9 CONCLUSION

Existing sanitizers for fuzzing have multiple inherent limitations. In particular, they conflict to each other and introduce significant slowdown. In this paper, we designed and implemented EXPOZZER, a comprehensive, efficient, and faithful bug exposer for fuzzers. EXPOZZER uses dual-execution to detect bugs that can trigger divergences in data-flow or control-flow thus are exploitable. EXPOZZER incorporates an efficient co-design for dual-execution and fuzzers and multiple techniques to ensure the effectiveness of bug exposing. To show the effectiveness and efficiency of EXPOZZER on detecting common categories of memory bugs, we conduct experiments of applying EXPOZZER to real world programs and also compare EXPOZZER with two major sanitizers: ASAN and MSAN. The results show that EXPOZZER is effective in detecting both known and new memory bugs, and outperforms the combination of ASAN and MSAN in fuzzing throughput by at a factor of 10.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful suggestions and comments. This work was supported in part by the NSF Awards CNS-1815621 and CNS-1931208. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF. Bowen Wang and Kangjie Lu are co-first authors.

REFERENCES

[1] Google, "OSS-fuzz - continuous fuzzing of open source software," 2017. [online]. Available: https://github.com/google/oss-fuzz

[2] N. Stephens *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. Netw. Distrib. Syst. Secur. Symp.* 2016, pp. 1–16.

[3] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–14.

[4] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 745–761.

[5] Y. Chen *et al.*, "Savior: Towards bug-driven hybrid testing," *IEEE Symp. Secur. Privacy*, San Francisco, CA, USA, pp. 1580–1596, 2020.

[6] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 711–725.

[7] S. Gan *et al.*, "Collafl: Path sensitive fuzzing," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 679–696.

[8] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 697–710.

[9] C. Lyu *et al.*, "MOPT: Optimized mutation scheduling for fuzzers," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1949–1966.

[10] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2329–2344.

[11] W. You *et al.*, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2139–2154.

[12] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, pp. 309–318.

[13] E. Stepanov and K. Serebryany, "Memorysanitizer: Fast detector of uninitialized memory use in C++," in *Proc. 13th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2015, pp. 46–55.

[14] Google, "Addresssanitizer." Accessed: May 19, 2021. [Online]. Available: https://github.com/google/sanitizers/wiki/AddressSanitizer

[15] LLVM, "Clang 10 documentation memorysanitizer," 2018. [Online]. Available: https://clang.llvm.org/docs/MemorySanitizer.html

[16] D. Song *et al.*, "SoK: Sanitizing for security," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 1275–1295.

[17] Google, "MSAN: False positive with libaio?." Accessed: May 19, 2021. [Online]. Available: https://github.com/google/sanitizers/issues/688

[18] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing mayhem on binary code,” in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 380–394.

[19] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, “AEG: Automatic exploit generation,” *Commun. ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[20] S.-K. Huang, M.-H. Huang, P.-Y. Huang, H.-L. Lu, and C.-W. Lai, “Software crash analysis for automatic exploit generation on binary programs,” *IEEE Trans. Rel.*, vol. 63, no. 1, pp. 270–289, Mar. 2014.

[21] Y. Chen and X. Xing, “Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1707–1722.

[22] Y. Wang *et al.*, “Revery: From proof-of-concept to exploitable,” in *Proc ACM SIGSAC Conf Comput Commun Secur*, 2018, pp. 1914–1927.

[23] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, “FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities,” in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 781–797.

[24] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, “Automatic generation of data-oriented exploits,” in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 177–192.

[25] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-oriented programming: On the expressiveness of non-control data attacks,” in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 969–986.

[26] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, “Non-control-data attacks are realistic threats,” in *Proc. USENIX Secur. Symp.*, 2005, pp. 177–191.

[27] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity,” in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 147–163.

[28] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble, “Deterministic process groups in dos,” in *Proc. Operating Syst. Des. Implementation*, 2010, pp. 177–192.

[29] K. Koning, H. Bos, and C. Giuffrida, “Secure and efficient multi-variant execution using hardware-assisted process virtualization,” in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2016, pp. 431–442.

[30] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida, “kMVX: Detecting kernel information leaks with multi-variant execution,” in *Proc. 24th Int Conf. Architect. Support Program. Lang. Operating Syst.*, 2019, pp. 559–572.

[31] S. Volckaert *et al.*, “Secure and efficient application monitoring and replication,” in *Proc. USENIX Ann. Tech. Conf.*, 2016, pp. 167–179.

[32] P. Hosek and C. Cadar, “Varan the unbelievable: An efficient n-version execution framework,” in *Proc. ACM SIGPLAN Notices*, 2015, pp. 339–353.

[33] A. Limaye and T. Adegbija, “A workload characterization of the spec cpu2017 benchmark suite,” in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2018, pp. 149–158.

[34] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *Proc. ACM SIGPLAN Notices*, 2007, pp. 89–100.

[35] B. Dolan-Gavitt *et al.*, “Lava: Large-scale automated vulnerability addition,” in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 110–121.

[36] “Binutils cves,” 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=binutils

[37] D. Mu *et al.*, “Understanding the reproducibility of crowd-reported security vulnerabilities,” in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 919–936.

[38] “Linuxflaw,” 2019. [Online]. Available: https://github.com/mudongliang/LinuxFlaw

[39] B. Cox *et al.*, “N-variant systems: A secretless framework for security through diversity,” in *Proc. USENIX Secur. Symp.*, 2006, pp. 105–120.

[40] C. Basile, Z. Kalbarczyk, and R. K. Iyer, “Active replication of multithreaded applications,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 5, pp. 448–465, May 2006.

[41] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere, “Ghumvee: Efficient, effective, and flexible replication,” in *Proc. Int. Symp. Found. Pract. Secur.*, 2012, pp. 261–277.

[42] F. B. Cohen, “Operating system protection through program evolution.” *Comput. Secur.*, vol. 12, no. 6, pp. 565–584, 1993.

[43] S. Forrest, A. Somayaji, and D. H. Ackley, “Building diverse computer systems,” in *Proc. 6th Workshop Hot Top. Operating Syst.*, 1997, pp. 67–72.

[44] V. Pappas, M. Polychronakis, and A. D. Keromytis, “Practical software diversification using in-place code randomization,” in *Moving Target Defense II*. New York, NY, USA: Springer, 2013.

[45] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, “SoK: Automated software diversity,” in *Proc. IEEE Symp. Secur. Privacy*, 2014, pp. 276–291.

[46] M. Cao *et al.*, “Different is good: Detecting the use of uninitialized variables through differential replay,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1883–1897.

[47] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, “Test-case reduction for c compiler bugs,” in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2012, pp. 335–346.

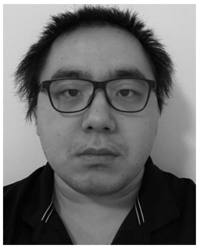
[48] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, “Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations,” in *Proc. IEEE Symp. Secur. Privacy*, 2014, pp. 114–129.

[49] Y. Chen and Z. Su, “Guided differential testing of certificate validation in SSL/TLS implementations,” in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 793–804.

[50] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, “Nezha: Efficient domain-independent differential testing,” in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 615–632.

[51] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, “Coverage-directed differential testing of JVM implementations,” in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2016, pp. 85–99.

[52] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov, “A security policy oracle: Detecting security holes using multiple API implementations,” *ACM SIGPLAN Notices*, vol. 46, no. 6 2011, pp. 343–354.



Bowen Wang is working toward the PhD degree at the Computer Science & Engineering Department, University of Minnesota. He is advised by Professor Stephen McCamant and his research interests include includes bug detection and system security.

Kangjie Lu received the PhD degree in computer science from the Georgia Institute of Technology. He is an assistant professor with the Computer Science & Engineering Department, University of Minnesota-Twin Cities. His current research nterests include aims to secure computer systems by hardening code and design, finding vulnerabilities, and detecting privacy leaks.

Qiushi Wu is working toward the PhD degree at the Computer Science & Engineering Department, University of Minnesota. He is advised by Professor Kangjie Lu and his research interests include systems security, patch analysis, and bug finding.

Aditya Pakki is working toward the PhD degree at the Computer Science & Engineering Department, University of Minnesota. He is advised by Professor Kangjie Lu and his research interests include systems security, operating systems, and bug finding.