Program Environment Fuzzing

Ruijie Meng
National University of Singapore
Singapore
ruijie_meng@u.nus.edu

Gregory J. Duck*
National University of Singapore
Singapore
gregory@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore
Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Computer programs are not executed in isolation, but rather interact with the execution environment which drives the program behaviors. Software validation methods thus need to capture the effect of possibly complex environmental interactions. Program environments may come from files, databases, configurations, network sockets, human-user interactions, and more. Conventional approaches for environment capture in symbolic execution and model checking employ environment modeling, which involves manual effort. In this paper, we take a different approach based on an extension of greybox fuzzing. Given a program, we first record all observed environmental interactions at the kernel/user-mode boundary in the form of system calls. Next, we replay the program under the original recorded interactions, but this time with selective mutations applied, in order to get the effect of different program environments—all without environment modeling. Via repeated (feedback-driven) mutations over a fuzzing campaign, we can search for program environments that induce crashing behaviors. Our $\mathcal{E}$FUZZ tool found 33 previously unknown bugs in well-known real-world protocol implementations and GUI applications. Many of these are security vulnerabilities and 16 CVEs were assigned.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

greybox fuzzing; program environment; software testing

ACM Reference Format:
Ruijie Meng, Gregory J. Duck, and Abhik Roychoudhury. 2024. Program Environment Fuzzing. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690229

1 INTRODUCTION

Computer programs are not executed in isolation, but rather interact with a complex *execution environment* which drives the program behaviors. Inputs received from the environment, such as configuration files, terminal input, human-user interactions, and network sockets, directly affect the internal program state which, in turn, governs how the program executes. Outputs sent to the environment, such as terminal output and sockets, provide useful clues that reflect these program states and behaviors. If the program is buggy, some environmental interactions may cause the program to crash or otherwise misbehave. *Fuzz testing* (or *fuzzing*) [7] is a widely-used automatic method that can find such program (mis)behavior. Ideally, fuzzing should be executed under different execution environments to comprehensively explore diverse program behaviors. However, capturing the effect of complex environments has always been a challenge for all program-checking methods—be it software verification, analysis, or testing. A dominant approach for handling different environments is *environment modeling*, which is used by verification and analysis methods including model checking and symbolic execution.

Algorithmic verification methods, such as model checking, conduct a search over the space of program states. Thus to verify an open software system interacting with the environment, model-checking methods typically describe the environment as a separate process. This process captures an *over-approximation* of possible behaviors that could be exhibited by real concrete environments. The environment process is then composed with the open software system, forming a closed system which can then be subjected to search. Environment synthesis for model checking has been studied in works such as [36]. These approaches depend on user-provided specifications to implement a *safe* approximation of the environment, and do not use concrete environments to demonstrate program errors.

In symbolic analysis methods [5, 11] and tools such as KLEE [10], environment capture is handled by redirecting calls to "environment models". These models are hand-written `C` code, specifically, the KLEE paper [10] mentions writing 2500 lines of `C` code to implement 40 system calls. Note that even these are simplified descriptions of the system calls. Although this approach is modeling-based, these works show a more direct attempt to handle program inputs from different environmental sources such as files, networks, etc.

In this paper, we take a fresh look at the problem of program environment capture, and provide a solution in the context of fuzz testing. Greybox fuzzing uses a biased random search over the domain of program inputs to find crashes and hangs. We aim to extend greybox fuzzing over the *full environment* without resorting to modeling. Our approach is to first run the program normally, but also to *record* all interactions between the program and environment that can be observed at the user/kernel-mode boundary (e.g., *system calls*). These interactions serve as the set of initial seeds. Next, the program is iteratively run again as part of a fuzzing loop, but this time *replaying* the original recorded interactions. During the replay, the fuzzer will opportunistically mutate the interactions recorded for system calls to observe the effect of environments different from that of the original recording. In effect, the program environment

*Joint first author

This work is licensed under a Creative Commons Attribution International 4.0 License.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690229

is fuzzed at the system call layer. Our approach does not conduct any abstraction of possible environments; it (implicitly) works in the space of real concrete environments.

We present a generic approach for fuzzing the full program environment. Existing greybox fuzzers are limited to fuzzing *specific* input sources, such as an input file specified by the command line (e.g., AFL [37] and AFL++ [17]), or a network socket over a specific network port (e.g., AFLNet [31] and Nyx-Net [34]). Our approach extends the scope of fuzzing to include *all* environmental inputs, meaning that any input is considered a fuzz target, regardless of source. We also propose a generic fuzzing algorithm to (implicitly) generate different program environments, thereby exploring diverse program behaviors. We have implemented our approach of program environment fuzzing in the form of a new greybox fuzzer called $\mathcal{E}$fuzz. We evaluate $\mathcal{E}$fuzz against two categories of user-mode programs under Linux: network protocol implementations and GUI applications, both of which are considered challenging subjects for existing fuzzers [7, 17]. In real-world and well-known applications, such as Vim and GNOME applications, $\mathcal{E}$fuzz found 33 previously unknown bugs (24 bugs confirmed by developers, which include 16 new CVEs). The bugs found include null-pointer dereferences, buffer overreads, buffer overwrites, use-after-frees, and bad frees, all triggered by fuzzing diverse environmental inputs including sockets, configuration files, resources, cached data, etc.

In summary, we make the following main contributions:

- We propose a new greybox fuzzing methodology to capture the effect of complex program environments—all without environment modeling or manual effort.
- We present a new fuzzing algorithm based on the full environmental record and replay at the user/kernel-mode boundary.
- We implemented the approach as a generic fuzzer ($\mathcal{E}$fuzz) capable of testing various program types, including two categories of recognized challenging subjects. In our evaluation, we found 33 previously unknown bugs and received 16 CVE IDs.

Our tool is publicly available at

https://github.com/GJDuck/EnvFuzz

2 BACKGROUND AND MOTIVATION

2.1 Motivating Example

As an initial motivating example, we consider a calculator application implemented using a *Graphical User Interface* (GUI). A human user makes inputs in the form of mouse movements, keystrokes, button presses, *etc.*, and the application reacts by generating outputs that update the graphical display. For example, by pressing the button sequence ⟨1, +, 2, =⟩, the application responds by displaying the answer (3).

Like all software, the calculator application may contain bugs, and these bugs can be discovered using automatic software testing methods such as fuzzing. For example, a fuzzer could apply the mutations (+)→(/) and (2)→(0) to construct a new button press sequence ⟨1, /, 0, =⟩ that will cause a crash (`SIGFPE`) if the calculator application were to not properly handle division by zero. Although most mutations will be benign (non-crashing), typical fuzzers mitigate this with a combination of high throughput (e.g., 100s of executions per second), program feedback (e.g., code coverage), and power scheduling (e.g., controlling mutation counts), increasing the likelihood of finding crashing inputs within a given time budget.

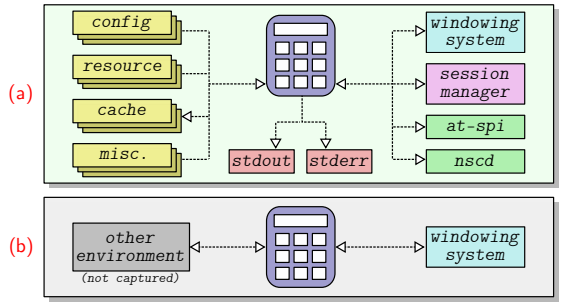

Figure 1: (a) is a calculator application with the *full* environment, including regular file I/O, standard streams, and socket/event fds to various system services. (b) is a *simplified* environment with a single input/output (windowing system socket), where all other interactions are not captured.

However, most existing greybox fuzzers, such as AFL [17, 37] and AFLNet [31], do not consider all input sources when producing mutated inputs. These fuzzers only target a specific class of inputs by default. For example, AFL only targets standard input (`stdin`) or a file specified by the command line. Similarly, AFLNet only targets network traffic over a specific port for a specific popular protocol (e.g., `ftp` and `smtp`). Essentially, these existing fuzzers use a simplified program environment, where program behaviors (and potential crashes) are driven by a single input source, and it is up to the tool user to decide *which* input source to fuzz. All other input sources are considered as "static", i.e., unmutated and unchanged between test cases. Furthermore, most existing fuzzers are specialized to specific *types* of inputs, such as regular files or popular network protocols.

In reality, most programs have a more complicated interaction with the environment beyond that of a single input source. For example, if we consider the `gnome-calcuator` application as part of the GNOME Desktop Environment for Linux. This application will open 706 distinct file descriptors under a minimal test (i.e., open and close the application window), including:

- 674× regular files, including configuration, cache, and GUI resources (icons/fonts/themes).
- 7× socket connections to the windowing system, session manager, and other services.
- Miscellaneous (e.g., special files, devices, and `stderr`).

The calculator application with a full environment is illustrated in Figure 1 (a).

2.2 Limitations of Conventional Fuzzing

Fuzzing requires two key decisions to be made before use:

- *Input Selection*: Which input should be fuzzed?
- *Environment Modelling*: How to handle other inputs?

For the button-press example, the fuzz target would be the windowing system socket over which button-press events are received.

Thus, for the purposes of fuzzing, we use a simplified environment as illustrated in Figure 1 (b). In the case of the calculator application, the simplified environment is somewhat naïve, since the target socket is only one of many possible input sources (706 possibilities). Consequently, only a small fraction of the actual environment is subjected to fuzzing. Assuming, for the sake of example, that the windowing system socket is selected. The next step is to choose a fuzzer. Since the input is a socket rather than a file, a network protocol fuzzer, such as AFLNet, will be suitable. AFLNet works by fuzzing inbound network messages and parsing the response codes from outbound messages as feedback to guide the fuzzing process. However, AFLNet only supports a limited set of pre-defined network protocols, and this does not include support for windowing system protocols. Even if the necessary protocol support is available, the environment beyond the fuzz target must still be handled.

One approach is to fix all the remaining environments as most existing fuzzers do, where the program is consistently checked within a single environment across test cases. Obviously, this approach limits the explored program behaviors. Moreover, in some cases, such as fuzzing the calculator application and other GUI applications, this approach is impractical for existing fuzzers. Handling regular file I/O is relatively straightforward since files can be read from disk for each executed test case, with outputs easily discarded (e.g., piped to `/dev/null`). However, a program can interact with more than one external service, such as session managers, service daemons, and even human users. In order to execute a single test case as part of the fuzzing process, the system-environment interactions would need to be "reset" for each individual test case—something known to be slow. For human-driven inputs, this also implies that a human-in-the-loop is necessary, since the fuzzer needs human interaction to proceed from one test execution to another.

Another approach is to build a *model* of possible environmental interactions. However, modeling is non-trivial. For example, each external service will typically use its own specialized protocol, and there can be an arbitrary number of services in the general case. Furthermore, any model would need to be *accurate*, as an invalid interaction may cause the test subject to terminate early due to an error condition, thus hindering reaching potential bug locations. Environment modeling is a known problem in the context of model checking and symbolic execution. Many existing works [6, 10] address it by modeling the environment *manually*. However, these approaches tend to be limited to specific problem domains and lack scalability for the general case.

2.3 Core Idea

We now describe our approach. We do not explicitly enumerate all possible environments in a search space and then navigate this very large search space. Our approach (below) is more implicit.

- *Input Selection*: All environmental inputs are fuzzed.
- *Environment Modelling*: Avoid modelling. The inputs are executed under a given environment and the effect of different environments is captured by mutating the environmental interactions represented by system calls.

For the calculator example, we consider all environmental inputs as fuzz targets regardless of *type*. Thus, various files (e.g., configuration, cache, and resource), sockets (e.g., those utilized by the windowing system), and any other input sources, are abstracted as generic *inputs* to subject fuzzing, eliminating the need for special handling. Since the whole environment is the fuzz target, any remaining residual environment is essentially eliminated, avoiding the need for additional modeling.

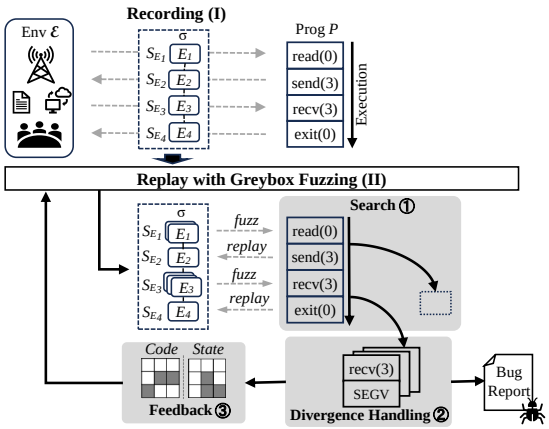

Figure 2: Overview of Program Environment Fuzzer $\mathcal{E}$FUZZ.

Building upon this concept, our approach first *records* all environmental interactions between the target program and its environment. Subsequently, the program is iteratively run again as part of a fuzzing loop, this time by *replaying* the interactions from the previous recording to substitute the original environment. Instead of replaying exactly the original recording, some of the interactions are *mutated* to implicitly generate the effect of different program environments, potentially uncovering new program behaviors.

To record the full environmental interactions, our approach works at the system call layer. This is motivated by the observation that most user-mode applications in Linux interact with the environment through the kernel/user-mode interface. For example, button presses and corresponding GUI updates flow through `recvmsg` and `sendmsg` systems calls over a socket. Similarly, pipes, streams and file I/O flow through standard `read` and `write` system calls. As a result, by recording system calls, we also record the full environmental interactions of the program, including system-environment interactions and human interactions, regardless of the underlying input type or the nature of the program.

3 SYSTEM OVERVIEW

Based on our core idea, we design a generic program-environment fuzzer using a *record and replay* methodology. This fuzzer, called $\mathcal{E}$FUZZ, is illustrated in Figure 2. At a high level, $\mathcal{E}$FUZZ consists of two phases: the *record* phase (Phase I) records the full interactions between the program-under-test P and its environment $\mathcal{E}$, and the *replay-with-greybox-fuzzing* phase (Phase II) replays and fuzzes the recorded interactions. Phase II captures the effect of different program environments and uncovers new program behaviors.

Phase I: Recording. In the *recording* phase, the program P is run normally within some test environment $\mathcal{E}$. The program interacts with the environment (files, sockets, human input, etc.) via a sequence of system calls, which are intercepted by $\mathcal{E}$FUZZ and saved into a *recording* σ. Here, σ is an in-order sequence of *records* (e.g.,

$\sigma = [E_1, E_2, E_3, E_4]$), where each record E stores all of the necessary details for reconstructing each corresponding environmental interaction in Phase II. These details include the system call number, system call arguments, buffer contents (if applicable), and the return value. These records are then saved into a respective seed corpus ($\mathcal{S}_E = \{E\}$) to serve as the initial seeds for the subsequent replay and greybox fuzzing.

Phase II: Replay with Greybox Fuzzing. Phase II combines environment replay with greybox fuzzing. The idea is two-fold:

(1) Faithfully *replay* the recorded environment interactions to reconstruct (deep) program states observed during Phase I;
(2) *Fuzz* each reconstructed state using greybox fuzzing.

Faithful replay works by re-running the program, but using the recording σ as a substitute for the original test environment $\mathcal{E}$. This again works by intercepting system calls, but this time the corresponding record ($E \in \sigma$) is *replayed* as a substitute for the real interaction.

To uncover different program behaviors for bug discovery, the core of $\mathcal{E}$FUZZ lies in greybox fuzzing. However, unlike traditional fuzzers, $\mathcal{E}$FUZZ works by fuzzing the recorded environmental interactions ($E \in \sigma$), rather than targeting specific files, as with AFL, or sockets, as with AFLNET. This works as follows: for each state reconstructed, $\mathcal{E}$FUZZ faithfully replays the next environmental interaction E in sequence from σ to advance state reconstruction. *In addition*, $\mathcal{E}$FUZZ selects seeds from the seed corpus $\mathcal{S}_E$, assigns energy, and introduces mutations to generate *mutant* interactions. Each mutant interaction is replayed in a *forked* branch of execution, where the program's behavior is observed (see Figure 2 ①).

Following the execution of a mutant interaction, the program behavior may *diverge* significantly from the original recording. Such divergence can include the program invoking different system calls, or invoking existing system calls but in a different order. For example, as shown in Figure 2, the `exit(0)` system call could be changed into `recv(3)`. Such behavior divergence presents a technical challenge for advancing replay, since only the original recording (σ) is available. Indeed, the main goal of fuzzing is to explore novel (divergent) program behaviors in order to discover bugs. To resolve this challenge, $\mathcal{E}$FUZZ introduces the notion of *relaxed* replay (as opposed to *faithful* replay) that is designed to progress divergent program execution after mutation (see Figure 2 ②).

At the end of each execution, similar to traditional greybox fuzzing, program feedback is used to determine *interesting* mutant interactions (see Figure 2 ③). The interesting interactions are saved into the seed corpus for future mutation. Additionally, mutations triggering program crashes are saved and reported to the user. The fuzzing campaign repeatedly iterates over reconstructed program states until a time budget is reached.

4 ENVIRONMENT FUZZING

We describe and explain the $\mathcal{E}$FUZZ algorithm in this section.

4.1 Environment Recording and Replay

For recording the environment, $\mathcal{E}$FUZZ implements a system call *interceptor routine* that acts as a proxy (i.e., "man-in-the-middle") between the program P and the kernel. Thus, when the program invokes a system call, such as a `read` or `write`, the call will be routed to the interceptor routine. The routine first *forwards* the system call to the underlying kernel and waits for the result. Once the underlying system call completes, the interceptor routine will then save relevant information about the system call into a record E, including: the system call number (e.g., `read` and `write`), arguments (e.g., file descriptor, buffer pointer, and buffer size), buffer contents (where applicable), current thread ID, and the return value. The system call result is then returned back to the program P, which continues executing as normal.

Each individual record E represents an interaction between the program P and its environment $\mathcal{E}$. During recording, each record is appended onto an in-order sequence σ, otherwise known as the *recording*, and is also saved into the respective seed corpora. The recording σ contains the information necessary to reconstruct all program states previously observed during the recording phase. For *faithful* replay, the program is run once more, but this time the interceptor routine instead *replays* (rather than forwards) the previously-recorded E. For fuzzing, the original record is replayed, but with one or more mutations applied first. Such mutations represent modified environmental interactions, and can change the program behavior.

We now use an example to illustrate this process. Suppose that during recording, the program P calls `read`(0, *buf*, 100), which is forwarded to the kernel, and the user enters "`quit\n`" into `stdin` (*fd*=0). The interceptor routine will record the returned buffer contents ("`quit\n`") and the returned value (=5 bytes read) into a record E. Then, during replay with greybox fuzzing:

- For *faithful* replay, the program P is re-run, and calls the same `read` system call as before. Instead of forwarding the system call to the kernel, the interceptor routine copies the previously recorded contents from E, copying "`quit\n`" into *buf* and returning 5. This causes the program's execution to proceed equivalently to the original recording.
- For fuzzing, the record E is first *mutated* before it is replayed. For example, the buffer contents could be mutated into "`quip\n`", and this will likely cause the program's behavior to diverge as if this were the original user interaction—possibly exposing new behaviors and bugs.

The mutation is applied to the buffer contents of *input* system calls (e.g., `read`) as this can affect the program behaviors and cause behavior divergence. Other system calls, that do not affect the program behaviors (e.g., `write`), will not be mutated. The combination of faithful replay, and replay with mutation, forms the basis of $\mathcal{E}$FUZZ's greybox fuzzing algorithm.

4.2 Reflections on Search Challenges

After the recording phase, $\mathcal{E}$FUZZ has collected a set of initial seeds representing real environment interactions. Using these as a basis, $\mathcal{E}$FUZZ employs greybox fuzzing to generate new interesting seeds representing interactions with new program environments—each with the potential to induce novel program behaviors. In designing an efficient algorithm for searching the program environment space, there are two main challenges: (i) *statefulness*: how to effectively explore deep program behaviors? (ii) *throughput*: how to maintain high fuzzing throughput?

```
int main()   {
   char buf[SIZE]; int r;
   r = read( 0, buf, SIZE); ...;
   r = send( 3, buf, r);    ...;
   r = recv( 3, buf, SIZE); ...;
   r = send( 3, buf, r);    ...;
   r = recv( 3, buf, SIZE); ...;
   r = write(1, buf, r);    ...;
   exit(0); }
```

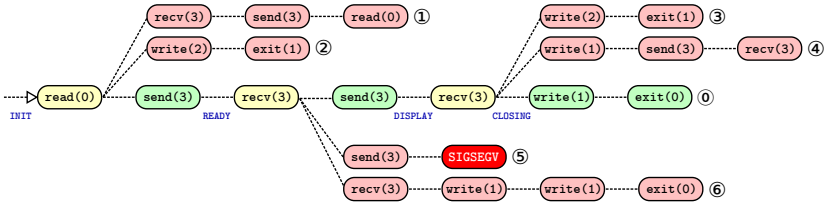

Figure 3: Illustration of the underlying fuzzing algorithm. Here, the example program reads from file descriptor 0, then interacts with socket (file descriptor 3). The fuzzer faithfully *replays* a previously *record*ed interaction ⓪, as well as several *mutant* interactions ①/②/③/④/⑤/⑥. Each mutant interaction is generated by mutating at least one input system call from the faithful replay. This causes the program's behavior to diverge, including exit with error ②/③, system call reordering ①/⑥, new I/O system call ④, and a crash ⑤. The program state {INIT, READY, DISPLAY, CLOSING} between select system calls is also illustrated.

Challenge (i): Statefulness. To better understand Challenge (i), we consider a calculator program that interacts over multiple input and output sources, as shown in Figure 3. The program begins in an INIT state, where it first parses a configuration file (file descriptor 0), then creates a user interface (GUI) by sending a message over the windowing system socket (file descriptor 3). The program then transitions into a READY state—i.e., waiting to accept mathematical expressions from the user interface. Subsequently, the program processes one user input expression (received from 3), and then sends the result back to the interface (send to 3), and the program transitions into the DISPLAY state. Finally, the user closes the interface (received from 3), and the program transitions into a CLOSING state. Here, the program writes a message to the terminal (file descriptor 1) before exiting. Our example is a simplification for brevity, as a real calculator program will typically interact with thousands of system calls, and may have many more internal states.

At the layer of system calls, the program is *stateful*, as it accepts a sequence of environmental inputs and adjusts its state accordingly. Some program behaviors are only reachable by specific states, which are in turn reachable only through specific input sequences. When fuzzing stateful programs, greybox fuzzers aim to exercise each observed state in order to explore the neighborhood of potential program behaviors, thereby having a greater chance to expose new bugs. However, state identification remains a challenging problem for fuzz testing in general. Existing works [3, 4, 31, 33] propose several heuristics for program state detection. For example, IJON requires states to be manually annotated, whereas AFLNet utilizes response codes from outbound messages to detect new states for well-known protocols. Either way, existing approaches require manual effort or are specialized to specific input sources.

We propose a generic approach that considers all input sources, such as files, sockets, and pipes, and consider how they affect program states. We consider each input system call as a *potential* state transition. For example, in Figure 3, after executing each input system call in sequence, the program transitions from the INIT to READY state, then from the READY to DISPLAY state, and finally from the DISPLAY to CLOSING state. Thus, each input can be fuzzed as a distinct transition between states, regardless of the input type (file, socket, etc.). However, some of the inputs may not trigger new transitions. This is mitigated by the power schedule [8], where inputs that fail to induce real state transitions are also less likely to expose new program behaviors observable via program *feedback*. As such, the corresponding input will be assigned less *energy*, and is naturally deprioritized for future mutations.

Challenge (ii): Throughput. To reach each observed state, $\mathcal{E}$FUZZ conducts a faithful replay of the recorded system calls. Upon reaching a state (i.e., before executing the corresponding input system call), $\mathcal{E}$FUZZ applies mutations to explore the neighboring program behaviors. If the fuzzer must always replay system calls from the root point, then multiple system calls need to be replayed to reach a specific state. For example, in the calculator example, a total of four system calls must be faithfully replayed to reach the DISPLAY state. This can significantly slow fuzzing throughput, especially for real-world examples where thousands of system calls may be required to reach a given state. To address this challenge, we propose a tree-based fuzzing algorithm that avoids (re)executing the same prefix sequence of system calls repeatedly. The algorithm is illustrated by the tree shown in Figure 3. Specifically, the original recording is faithfully replayed (without mutation), forming the "spine" of the tree, which is represented by the middle trace ⓪. Upon reaching an input system call, $\mathcal{E}$FUZZ additionally forks-off some number of *mutant* traces, creating the "branches" of the tree (e.g., ①/②). Each branch starts by replaying an original input with one or more *mutation* operators applied, and may involve further mutations of subsequent inputs. After executing each branch, $\mathcal{E}$FUZZ continues the faithful replay to grow the spine until the next input point, after which $\mathcal{E}$FUZZ forks-off more branches (e.g., ⑤/⑥). The process repeats once more (e.g., ③/④) before ⓪ terminates.

4.3 Fuzzing Search Algorithm

Based on the environment recording and replay technique, along with the efficient search strategy, we introduce a novel environment fuzzing algorithm, illustrated in Algorithm 1. The recording is shown in line 2 of Algorithm 1. After the recording, the program is executed normally, but with the interceptor routine FuzzSyscall replacing the standard system call interface (line 8). There are two main cases to consider: the replay is in the spine or in a branch (e.g., see Figure 3), and the program starts with running in the spine (*isBranch*←*false*). For the spine of the tree (line 12-line 25), $\mathcal{E}$FUZZ retrieves the next record E to be processed (line 13). For non-input system calls (e.g., write), the original record E is faithfully replayed "as-is" (line 14). Conversely, all input system calls (e.g., read) are treated as potential fuzzing targets, and a greybox fuzzing algorithm is used (line 15-line 24). Specifically, for each record E

Algorithm 1: Program Environment Fuzzing Algorithm.

```
Input  : Program P, environment interaction ℰ
Output : Crashing events C✗
Globals: Input-specific corpora S_E
1  func EnvFuzz(P, ℰ):
2      σ ← Record(P, ℰ)                                  ▷ Recording
3      for E ∈ σ do S_E ← {E}
4      repeat
5          FuzzReplay(P, σ)
6      until timeout reached or abort
7  func FuzzReplay(P, σ):                                ▷ Replay with Fuzzing
8      exec(P_[replace syscall with FuzzSyscall, isBranch←false], σ)
9  func FuzzSyscall(e):                                  ▷ Tree-based Search
10     if isBranch then
11         return EmulateSyscall(e, σ)                   ▷ Divergence Handling
12     else /* if isSpine then */
13         E ← head(σ); σ ← tail(σ)
14         if ¬isInput(e) then return ReplaySyscall(E)
15         for E′ ∈ S_E, i ∈ 1..energy(E′) do
16             E″ ← mutate(E′)
17             pid ← fork()
18             if pid = 0 then                           ▷ In child:
19                 isBranch ← true
20                 return ReplaySyscall(E″)
21             else                                      ▷ In parent:
22                 waitpid(pid, &status)
23                 if isCrash(status) then add E″ to C✗
24                 if isInteresting(E″) then add E″ to S_E
25         return ReplaySyscall(E)                       ▷ Grow Spine
```

corresponding to the input syscall, ℰFuzz will iterate over each seed E' from corpus $\mathcal{S}_E$. For each E', ℰFuzz applies one or more standard *mutation operators*, to further mutate the input buffer contents, and thereby generating a new seed E'' (line 16). The current implementation uses mutation operators from other fuzzers, e.g., `havoc` from AFL [31, 37]. The number of mutations is controlled by a power schedule (*energy*) (line 15).

To execute the new seed E'', the algorithm first forks the program into a *parent* and *child* process (line 17). The seed E'' is executed in the child, forming a *branch* of the tree (e.g., see Figure 3), while the parent waits for the child's termination (line 22). After applying a mutation in the child, the interceptor routine `FuzzSyscall` processes the subsequent system calls using a different method (line 11), which will be discussed in Section 5. Following the termination of the child, the parent examines the result. Crashing mutations are saved into a special corpus $C_\times$ that forms the output of Algorithm 1 (line 23). Otherwise, the fuzzing feedback (discussed in Section 4.4) is used to determine whether the mutated seeds are *interesting* or not, and interesting seeds are saved into $\mathcal{S}_E$ for future mutation see line 24 ; the decision on whether a seed is interesting or not, is conducted based on fuzzing feedback which is discussed in the next subsection. Subsequently, ℰFuzz grows the spine by continuing faithful replay (line 25). After the fuzzing campaign is complete, the ℰFuzz infrastructure also supports replaying any of the $C_\times$ corpus to reproduce discovered bugs.

An illustration of this fuzzing algorithm on a simple example program appeared in Figure 3.

4.4 Fuzzing Feedback

Greybox fuzzing relies on feedback to select "interesting" seeds (line 24 in Algorithm 1) to guide the search towards novel program behaviors, thereby increasing the likelihood of discovering bugs [21]. A common form of feedback is *branch coverage*, as used by many modern fuzzers [17, 37]. Here, seeds that cover new branches (code paths) have the potential to explore different behaviors, and thus are considered interesting and saved into the corpus for future mutation. Most fuzzers collect branch coverage feedback using compiler instrumentation (e.g., `afl-gcc`). Instrumentation can also be inserted directly into binary code using *static binary rewriting*, such as with E9AFL [18]. ℰFuzz supports branch coverage feedback and operates directly on binaries to maximize generality.

For the case of stateful programs, branch coverage alone is generally considered insufficient [3, 31]. As such, *state feedback* has been proposed in collaboration with branch coverage to guide the fuzzing process. Here, seeds that cover new state transitions are also considered "interesting" and are similarly added to the corpus. However, as discussed in Section 4.3, automatically inferring program states is challenging, especially for binary code. Our approach is to treat each input message as a *potential* state transition. We leverage program *outputs* (e.g., `write`) as a proxy for detecting states. Our heuristic is that, under certain inputs, a program will generate output that is contingent on its internal states, and thus outputs can provide insights into these states. To mitigate the impact of outputs with unknown structures/formats, we employ locality-sensitive hashing and clustering based on the *Hamming distance* [16, 26]. ℰFuzz can utilize both branch and state feedback to guide the search.

5 RELAXED REPLAY FOR DIVERGENCE

After a mutated input is replayed in a branch, it is common for the program's behavior to *diverge* from the original recording, as illustrated by the branches ①,...,⑥ in Figure 3. Divergence could include: exiting with error ②/③, system call reordering ①/⑥, new system call invoking ④, or even the program crashing ⑤. For example, suppose the last input from Figure 3 receives a command "`quit\n`" from the socket, causing the program to enter the `CLOSING` state and exit. However, mutant replay could change the command to "`quip\n`", foiling the state transition, and causing the program's behavior to diverge from the original recording.

This poses a challenge that is described as follows. During the recording phase, ℰFuzz will construct an in-order sequence of records σ. Assuming that $\sigma = [\sigma_1, E, \sigma_2]$, where E is an input, then during the fuzzing phase, ℰFuzz faithfully replays the prefix σ_1 (as part of the spine) before reaching E. Next, ℰFuzz mutates E to generate one (or more) mutant E', after which E' is replayed as a substitute for E. After replaying E', the faithful replay of σ_2 may no longer be possible due to program behavior divergence, i.e., the mutant sequence $\sigma' = [\sigma_1, E', \sigma_2]$ may be *infeasible*. The problem is that ℰFuzz only has the original recording σ to work with.

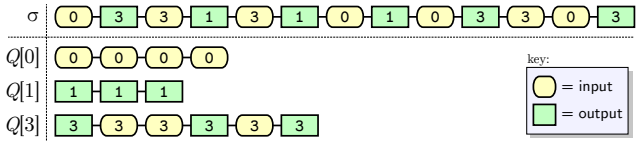

Figure 4: Illustration of the *global* ordering (σ) for faithful replay and a *local* ordering (Q) for relaxed replay. The relaxed replay partitions σ into a set of miniqueues ($Q[fd]$) indexed by the file descriptor, each of which defines a local ordering specific to each *fd*.

To address this problem, we introduce the notion of *relaxed replay*. The key idea is to use system call *emulation* (line 11 in Algorithm 1) to construct continuations of program execution that diverge from σ. Relaxed replay uses a set of *emulation routines*, one for each syscall number, where each routine takes the syscall arguments and returns a result (i.e., return value, buffer contents) on a "best-effort" using available information. Unlike faithful replay, these routines can be called at any time and in any order, and do not necessarily need to follow the original recorded system call ordering. Crucially, the emulation routines should only return *plausible* results—i.e., there exists a real (modified) environment $\mathcal{E}'$ from which the result could occur. Plausibility is necessary to avoid *false positives*—i.e., crash reports that are irreproducible under any real environment. For plausibility, we generalize assumptions used by existing fuzzers, wherein any I/O modification (e.g., in AFL) and reordering (e.g., in AFLNet) are considered plausible. We now discuss these two cases in detail.

5.1 Relaxing I/O System Call Ordering

After mutation, programs often invoke I/O system calls in a different order from that of the original recording. To handle this case, our approach is to first *partition* σ into a set of *miniqueues* $Q[fd]$, with one miniqueue specific to each I/O source (i.e., *file descriptor*, *fd*). The approach is illustrated by example in Figure 4. Here, under the *global ordering* (σ) for faithful replay, only a read system call from file descriptor 0 can be serviced. However, after mutation, the program may attempt I/O on a different file descriptor. To handle such cases, our approach allows I/O system calls to be directly serviced from the corresponding miniqueue $Q[fd]$ under a *local ordering* specific to each *fd*, rather than the original global ordering (σ). The partitioning and local ordering is plausible under the assumption that I/O system calls can be reordered.

It is also common for programs to use the `poll` system call[1] to query which I/O operations are currently possible. Relaxed replay must also handle the poll system call using emulation. The algorithm is shown in Algorithm 2, and is a concrete example of an emulation routine. Here, poll is emulated based on the *current* state of Q (line 3-line 12) and returning:
(i) *End-of-file* (`POLLHUP`) for an empty miniqueue (line 7);
(ii) *input ready* (`POLLIN`) or *output ready* (`POLLOUT`) if the queue head matches the requested event (line 9-line 10);
(iii) `0x0` (a.k.a. no event) otherwise.

[1]See the `poll` manpage for more information.

Algorithm 2: Emulated `poll` routine.

Input : Array of `pollfd` structs, Q derived from σ
Output : Number of non-zero *revents*

```
1  func EmulatePoll(fds, Q):
2      while true do
3          r ← 0; h ← 0
4          for i ∈ 0..|fds|−1 do
5              E ← head(Q[fds[i].fd])
6              if E = EOF then
7                  fds[i].revents = POLLHUP; h++
8              else
9                  fds[i].revents = fds[i].events &
10                     (isInput(E)? POLLIN: POLLOUT)
11                 r += (fds[i].revents? 1: 0)
12         if r > 0 ∨ h > 0 then return r
13         fd ← pick(fds, Q); Q[fd] ← reorder(Q[fd])
```

If at least one of the returned events is non-zero, then the poll operation successfully completes (line 12) and execution continues. Otherwise, the poll operation will block. To avoid blocking, the algorithm heuristically *picks* a file descriptor and *reorders* the corresponding miniqueue (line 13), allowing Algorithm 2 to always terminate (without blocking) in the next iteration of the outer-loop.

5.2 Relaxing I/O System Calls

Input system calls are emulated by an implicit `poll` operation, followed by popping the corresponding miniqueue $Q[fd]$. The popped record is replayed, possibly subject to further mutation. If the implicit poll operation indicates the miniqueue is empty (`POLLHUP`), the input system call returns 0 indicating an end-of-file (`EOF`).

Emulated output system calls similarly pop the corresponding miniqueue, but always succeed even if the queue is empty. This handles the common case where a mutation causes the program to generate additional output, such as a warning or error message that is not present in the original recording σ. Modified or extraneous outputs can generally be ignored, as outputs do not affect the program behavior. However, outputs do provide useful hints about the program state, which is used as fuzzing feedback.

5.3 Relaxing Non-I/O System Calls

Other system calls are handled using heuristics, such as:

- *Emulate*: emulate (plausible) effects of the system call;
- *Forward*: pass the system call "as-is" to the underlying O/S;
- *Fail*: fail the system call with an error condition (e.g., `ENOSYS`).
- *Exit*: as a last resort, terminate the branch with `exit`.

In addition to I/O system calls, $\mathcal{E}$fuzz also implements several specialized emulation routines for other common system calls, including time (e.g., `clock_gettime`/etc.) and thread-related (e.g., `futex`/`clone`/etc.) system calls. For example, time-related system calls are handled by emulating a global *monotonically increasing* clock t. The clock t is first initialized to the last time observed in the recording before the branch, and t is then incremented for each subsequent emulated system call after the branch. This ensures that

emulated time-related system calls always return *plausible* results—i.e., the system time always flows forwards. The system calls related to memory management (e.g., `brk`/`mprotect`/`madvise`/etc.) are generally *forwarded* to the underlying O/S "as-is" without special handling.

Sometimes neither system call emulation nor forwarding is applicable. For example, due to behavior divergence, the program may attempt to access a file that does not appear in the original recording (σ). As such, $\mathcal{E}$FUZZ has no information about the file contents, or whether the file even exists. In such cases, relaxed replay can *fail* the system call with an error (e.g., `ENOSYS`), allowing for execution to proceed and giving the program a chance to recover. As a last resort, relaxed replay will exit the program if no other alternative is possible. This occurs when a program ignores failure, e.g., by re-invoking the same system call again in a loop.

6 IMPLEMENTATION

We implemented the approach of $\mathcal{E}$FUZZ as a generic program environment fuzzer that can handle a diverse range of user-mode Linux applications, including GUI applications and network servers. $\mathcal{E}$FUZZ is built on top of a full environment record and replay infrastructure, similar to that of `rr`-debug [29]. In total, the $\mathcal{E}$FUZZ toolchain is implemented in over ~13k source lines of `C++` code.

The recording phase records all information that is necessary to faithfully replay the program P during fuzzing. In addition to system calls (the main focus of our discussion), the recording also includes additional information, such as the command-line arguments, environment variables, signals, thread interleavings, and special non-deterministic instructions (e.g., `rdtsc`). System call interception is implemented using a variety of techniques. The common case is handled using *static binary rewriting* to rewrite the `syscall` instruction in `libc`, which diverts control-flow to the framework's interceptor routine. For this, we use the E9PATCH [14] binary rewriting system. In addition, the framework also rewrites the *virtual Dynamic Shared Object* (vDSO) at runtime, and also uses `seccomp` to generate a signal (`SIGSYS`) that is used to intercept system calls outside of `libc` (less common case). Our framework does not use `ptrace`, and thus avoids kernel/user-modes switches during replay for the common case.

Multi-threaded programs are handled by *serializing* system calls during the recording phase, meaning that only one thread will run at a given time. The recording phase runs the program normally using serialized system threads, whereas the replay-with-fuzzing phase uses lightweight *fibers* as a replacement of system threads. This design avoids one of the main technical limitations of `fork()`, namely, that only the callee system thread will actually be cloned during a fork operation.[2] In contrast, fibers are threads of execution that are implemented purely in user-mode, and where context switching is determined by the recorded schedule (σ). Since there is no user-kernel interaction during replay, fibers can be used as a drop-in replacement of system threads without special handling. Furthermore, since fibers are purely implemented in user-mode, they survive the fork operation intact, which is necessary for the $\mathcal{E}$FUZZ fuzzing algorithm.

$\mathcal{E}$FUZZ is also designed to operate directly on binaries without the need for source code. $\mathcal{E}$FUZZ uses both state and branch coverage as feedback. To collect the branch feedback, binaries can be instrumented using a modified version of E9AFL [18]. State coverage feedback does not require instrumentation. Our implementation can record and fuzz large applications, including the subjects listed in our evaluation.

7 EVALUATION

To evaluate the effectiveness of $\mathcal{E}$FUZZ, we seek to answer the following research questions:

RQ.1 **New bugs.** Can $\mathcal{E}$FUZZ find previously unknown bugs in real-world and widely-used programs? Is fuzzing the program environment necessary to reveal these bugs?

RQ.2 **Comparisons.** How many additional bugs does $\mathcal{E}$FUZZ discover over the baseline? How much more code coverage does $\mathcal{E}$FUZZ achieve compared to the baseline? Are the additional bugs and code coverage improvements related to program environment fuzzing? How many tests can $\mathcal{E}$FUZZ execute per second compared to the baseline?

RQ.3 **Ablations.** What is the impact of each component on the performance of $\mathcal{E}$FUZZ?

7.1 Experiment Setup

Subject Programs. $\mathcal{E}$FUZZ is a generic fuzzer capable of testing a broad spectrum of user-mode programs in Linux. Given the scope of applications that $\mathcal{E}$FUZZ can fuzz, we shall focus on two core categories of program: network protocols and *(Graphical) User Interface* GUI/UI applications that interact with a human user via the windowing system or terminal. These two categories have been recognized as challenging for fuzzing [7]. For example, fuzzing GUI applications with AFL++ [1] is "*not possible without modifying the source code*".[3] Since $\mathcal{E}$FUZZ works at the abstraction of the kernel/user-mode boundary, it can fuzz GUI applications and other difficult subjects without special handling. By targeting challenging fuzzing targets, we aim to demonstrate the generality of $\mathcal{E}$FUZZ.

In total, we collect 20 subjects as detailed in Table 1. For network protocols, we collect subjects from PROFUZZBENCH [27], a widely-used benchmarking platform for evaluating the network-enabled fuzzers. However, for GUI applications under Linux, there is no existing fuzzing dataset. We therefore select subjects from frequently-used and well-known applications and frameworks, including text editors (UI), visual shells (UI), GNOME desktop environment (GUI), Qt (GUI), and the underlying windowing system (GUI).

Comparisons. To the best of our knowledge, no existing fuzzers target the full program environment. In the realm of fuzzing network protocols, AFLNET is the first network fuzzer, and also recommended by AFL++ for fuzzing network services. NYX-NET enhances the fuzzing throughput of AFLNET by introducing innovative hypervisor-based snapshots. Unfortunately, we cannot compare with AFL++ since it does not work on many of the network protocols. As shown in the paper of NYX-NET [34], AFL++ only works on 5 of the PROFUZZBENCH subjects. Furthermore, for these 5 subjects,

[2] See the `fork` manpage for more information.

[3] https://aflplus.plus/docs/best_practices/#fuzzing-a-gui-program (as of writing).

Table 1: Subject programs used in the evaluation.

	Subject	Version		Subject	Version
Network Protocols	DCMTK	8326435	GUI & UI Applications	Gnome editor (gedit)	v41.0
	DNSmasq	b676923		Gnome Calculator	v42.9
	Exim	5a8fc07		Gnome System Monitor	v42.0
	Kamailio	2e2217b		Glxgears	v23.0.4
	Live555	2c92a57		Midnight Commander (MC)	v4.8.27
	OpenSSH	7cfea58		nano	v6.2
	OpenSSL	a7e9928		Vim	v8.2
	ProFTPD	7892434		Wireshark	v3.6.2
	Pure-FTPd	3296864		Xcalc	v1.8.6
	TinyDTLS	0e865aa		Xpdf	v3.04

AFL++ performs significantly worse than both AFLNet and Nyx-Net. Therefore, for network protocol subjects, we use AFLNet and Nyx-Net as baselines for comparison. AFL++ and AFL++-based fuzzers are also not able to fuzz GUI applications in Linux with user interactions [1]. Recent work [20] uses *test harness* generation to enable GUI fuzzing, but only for Windows applications. As such, there is no available fuzzer to compare against GUI applications under Linux.

Performance Metrics. We evaluate the performance of $\mathcal{E}$fuzz based on three primary metrics: *bug-finding capability*, *code coverage*, and *fuzzing throughput*. As recommended by the fuzzing community [9, 21], the ultimate metric of a fuzzer is the number of distinct bugs found. Since a fuzzer cannot find bugs in uncovered code, code coverage is important too, and thus serves as a secondary metric. While fuzzing throughput is not a mandatory evaluation metric, it may affect the efficacy of a fuzzer. We also report throughput to demonstrate the robustness of the fuzzer.

Experimental Infrastructure. All experiments were conducted on an Intel® Xeon® Platinum 8468V CPU with 192 logical cores clocked at 2.70GHz, 512GB of memory, and running Ubuntu 22.04.3 LTS. Each experiment runs for 24 hours. We report the average over 10 runs to mitigate the impact of randomness.

7.2 Discovering New Bugs (RQ.1)

Method. We ran $\mathcal{E}$fuzz on the subjects listed in Table 1 to discover bugs. We utilized the same bug oracles as traditional fuzzers (e.g., AFLNet and Nyx-Net), including crashes, hangs, assertion failures, and sanitizer violations. For initiating the fuzz campaign, we used initial seeds provided by the programs if available; otherwise, we provided standard user inputs as initial seeds. In the case of network protocols, we utilized their clients to send request messages. For GUI applications, we simulated typical user interactions; as an example, with a calculator, the application is opened to perform a simple addition calculation before it is closed. All inputs represent normal usage scenarios encountered in the real world. We subjected each program to a *24-hour run* (typical recommended length of a fuzz campaign [21]) to identify bugs. Upon finding bugs, we reported them to the developers for confirmation. In the case of bugs with potential security implications, we requested CVE IDs from the CVE Numbering Authority. All activities were conducted in a one-month period, including bug finding, debugging, reporting to developers, and requesting CVEs.

Results. Table 2 shows the distinct and previously unknown bugs found by $\mathcal{E}$fuzz. In the *Bug Description* column, we elucidate the root causes responsible for these bugs, and illustrate the immediate environmental factors in the *Environment* column. It is important to note that triggering a bug often requires hundreds of diverse environmental inputs. Therefore, we only listed the most relevant environmental input that exposed the bugs after mutation. Furthermore, we provide details about the bug types and their current status in the last two columns.

In total, we discovered 33 previously unknown bugs, out of which 24 have received confirmation from their respective developers. Developers had fixed 16 of these bugs by the time of paper submission. 16/24 bugs have been assigned CVE IDs. These bugs span various categories, including buffer overflows, use-after-frees, null pointer dereferences, and arithmetic exceptions. Furthermore, these bugs are triggered by fuzzing a diverse range of environmental inputs, including sockets, configuration files, multiple types of resource files, cached data, etc. Therefore, a fuzzer that exclusively concentrates on a singular input cannot expose all of these bugs.

These results highlight the significant bug-finding capability of $\mathcal{E}$fuzz. Moreover, they demonstrate the importance of program environment fuzzing, and $\mathcal{E}$fuzz has shown its effectiveness in this regard. Two case studies below illustrate bugs discovered by $\mathcal{E}$fuzz.

Case study: GNOME Desktop Environment. GNOME client applications (e.g., `gnome-calculator`, etc.) interact with the windowing system and several other services (Figure 1). $\mathcal{E}$fuzz is able to expose several bugs in multiple different input sources, including several bugs related to the windowing system and client libraries, bugs in the DBus socket connection to the session manager, as well as bugs in non-socket inputs (`loaders.cache`, `gtk.css`, etc.). As an example, we can consider Bug #12, which affects the `XESetWireToEvent()` function from `libX11`. This function fails to check whether the event values are within the bounds of the arrays that the functions write to. Instead, the function directly uses the value as an array index, leading to an intra-object overwrite and probable crash. This bug stems from the implicit trust that `libX11` functions place in the values supplied by an X server, following X11 protocol. However, the environment cannot be fully trusted, as a malicious server or proxy can impact applications. This bug was assigned CVE ID by X11 developers and received a CVSS score HIGH 7.5. We note that other subjects, including many GNOME applications, are also affected by this bug.

Case study: Bug #23 in GNU nano. GNU nano is a text editor for Unix-like operating systems and is part of the GNU Project. This bug appears in `read_the_list()` of `browser.c`. This function initiates an initial iteration over a directory using `readdir()` to obtain the current entries, followed by a rewinding action using `rewinddir()` to cache these entries. Subsequently, a second iteration employing `readdir()` is performed to directly access these cached entries. Unfortunately, before this second iteration, there is no boundary-checking mechanism. As a result, any environmental changes, such as directory deletions, can easily trigger a crash during the second iteration. This is precisely how $\mathcal{E}$fuzz exposes it. This bug existed from the first version of GNU nano in 2005 and had been hidden for 18 years!

Table 2: Statistics of bugs discovered by $\mathcal{E}$FUZZ; a total of 33 previously unknown bugs found, 24 bugs confirmed by developers, 16 bugs assigned CVE IDs, and 16 bugs fixed. (Note that, each color represents a distinct category of applications)

ID	Subject	Bug Description	Environment	Bug Type	Bug Status
1	Dcmtk	Failed to check bounds of stored `dicom.dic` data	Cached data	Buffer overflow	CVE-requested, fixed
2	Exim	Failed to check bounds of a corrupted `resolv.conf`	Configuration	Buffer overflow	Reported
3	Exim	Glibc failed to handle an empty `passwd` line	Special file	Null pointer dereference	Reported
4	Kamailio	Improperly handle a corrupted client request	Socket	Null pointer dereference	Reported
5	Live555	Improperly handle a malicious `SETUP` client request	Socket	Heap use after free	CVE-granted, fixed
6	Live555	Failed to check bounds of a corrupted `test.mkv`	Media resource	Buffer overflow	Reported
7	OpenSSH	Improperly handle a corrupted `sshd_config`	Configuration	Null pointer dereference	CVE-requested, fixed
8	OpenSSH	Improperly handle a corrupted `gai.conf`	Configuration	Null pointer dereference	Reported
9	Pure-FTPd	Glibc failed to handle a corrupted `timezone` file	Time resource	Null pointer dereference	Reported
10	gedit	Improperly handle a null value in `parse_settings()`	Configuration	Null pointer dereference	CVE-granted
11	gedit	Improperly handle a null value from `XRRGetCrtcInfo()`	Socket	Null pointer dereference	CVE-granted
12	Calculator	Failed to check bounds of requests, events and error IDs	Socket	Buffer overflow	CVE-granted, fixed
13	Calculator	Failed to check null value from `XIQueryDevice()`	Socket	Null pointer dereference	CVE-granted
14	Calculator	Improperly handle a corrupt `DBUS` message	Socket	Null pointer dereference	CVE-requested, fixed
15	Monitor	Improperly handle corrupted `loaders.cache`	Cached data	Bad free	CVE-granted
16	Monitor	Failed to handle a corrupted `gtk.css`	Theme resource	Null pointer dereference	CVE-requested, fixed
17	Glxgears	Failed to check bounds of `numAttribs` in messages	Socket	Buffer overflow	CVE-granted
18	Glxgears	Failed to check bounds of the string length	Socket	Buffer overflow	CVE-granted
19	MC	Failed to handle a corrupted `terminfo`	Configuration	Null pointer dereference	CVE-granted
20	MC	Improperly handle a corrupted `xterm-256color`	Configuration	Arithmetic exception	CVE-granted
21	MC	Improperly process error handler of `x_error_handler()`	Socket	Null pointer dereference	CVE-granted
22	nano	Failed to handle a corrupted `xterm` file	Configuration	Null pointer dereference	Reported
23	nano	Failed to check the inconsistent directory in disk	Cached data	Null pointer dereference	CVE-granted, fixed
24	Vim	Failed to handle a corrupted `xterm-256color`	Configuration	Null pointer dereference	CVE-granted
25	Vim	Failed to handle a corrupted `viminfo` file	Cached data	Null pointer dereference	CVE-granted, fixed
26	Wireshark	Failed to check null pointer in `initializeAllAtoms()`	Socket	Null pointer dereference	CVE-granted, fixed
27	Xcalc	Failed to handle null pointer from `XOpenDisplay()`	Socket	Null pointer dereference	CVE-requested, fixed
28	Xcalc	Failed to check write boundary in `_XkbReadKeySyms()`	Socket	Out-of-bounds write	CVE-granted, fixed
29	Xcalc	Failed to check read boundary in `_XUpdateAtomCache()`	Cached data	Out-of-bounds read	CVE-requested, fixed
30	Xpdf	Improperly handle invalid and corrupted locale data	Configuration	Null pointer dereference	Reported
31	Xpdf	Improperly handle invalid paper size in configuration	Configuration	Null pointer dereference	Reported
32	Xpdf	Failed to check pointer boundary returned from response	Socket	Bad free	CVE-requested, fixed
33	Xpdf	Failed to check array boundary returned from X server	Socket	Out-of-bounds read	CVE-requested, fixed

$\mathcal{E}$FUZZ discovered 33 previously unknown bugs in widely used network protocols and GUI applications, with 24 confirmed and 16 fixed by their developers. 16 of them were assigned CVEs.

7.3 Comparisons with Baselines (RQ.2)

Method. For network protocols, we compare $\mathcal{E}$FUZZ against two baselines AFLNET and NYX-NET under three aspects: the number of bugs found, code coverage, and fuzzing throughput. We omit GUI programs due to the lack of a suitable baseline. We configure all fuzzers employing the same initial seeds obtained from PROFUZZBENCH. Our evaluation of code coverage focuses on measuring branch coverage achieved on binaries. We utilize the original scripts provided by PROFUZZBENCH, to collect code coverage data and present their trends over time. We report the total number of bugs found, the average coverage, and the average fuzzing throughput achieved by each fuzzer across 10 runs of 24 hours.

Comparing Results on Bug Finding. Table 3 shows the total number of unique bugs found by each fuzzer. In all subjects, $\mathcal{E}$FUZZ discovered a total of 9 unique bugs, as detailed in Table 2. However, both AFLNET and NYX-NET could only find 2 (i.e., Bug #4 and Bug #5 in Table 2); in addition, neither fuzzer found any additional bug. The remaining 7 bugs were exposed by fuzzing non-socket environment inputs, such as cached data and resources. Since these environment inputs are not fuzzing targets for AFLNET and NYX-NET, they were unable to expose them. Furthermore, regarding bugs induced by network sockets, $\mathcal{E}$FUZZ successfully exposed the same number as AFLNET and NYX-NET. This demonstrates that $\mathcal{E}$FUZZ maintains the effectiveness in fuzzing a single environment source, although it fuzzes all environment sources.

Table 3: Number of unique bugs found by AFLNET, NYX-NET and $\mathcal{E}$FUZZ on subjects of network protocols.

Fuzzer	AFLNET	NYX-NET	$\mathcal{E}$FUZZ
#Bug	2	2	9

In the aspect of bug finding, $\mathcal{E}$FUZZ discovered 9 previously unknown bugs, while AFLNET and NYX-NET only discovered 2 without any additional bug found.

Comparing Results on Code Coverage. Figure 5 illustrates trends in average code coverage over time for AFLNET, NYX-NET and $\mathcal{E}$FUZZ. Across all subjects, $\mathcal{E}$FUZZ significantly outperformed

Table 4: Average branch coverage across 10 runs of 24 hours achieved by $\mathcal{E}$FUZZ compared to AFLNET and NYX-NET.

Subject	$\mathcal{E}$FUZZ	Compare with AFLNET			Compare with NYX-NET		
		Coverage	Improv	$\hat{A}_{12}$	Coverage	Improv	$\hat{A}_{12}$
DCMTK	15181.7	7564.9	+100.69%	1.00	9362.0	+62.16%	1.00
DNSmasq	8090.9	4066.7	+98.95%	1.00	4009.0	+101.82%	1.00
Exim	5642.7	4594.4	+22.82%	1.00	4935.2	+14.34%	1.00
Kamailio	23425.6	13466.1	+73.96%	1.00	17960.0	+30.43%	1.00
Live555	14319.0	10379.5	+37.95%	1.00	11436.0	+25.21%	1.00
OpenSSH	8584.5	7920.0	+8.39%	1.00	7631.5	+12.49%	1.00
OpenSSL	26225.9	19820.4	+32.32%	1.00	25330.1	+3.54%	1.00
ProFTPD	19478.0	17654.0	+10.33%	1.00	16504.0	+18.02%	1.00
Pure-FTPd	7182.8	5309.0	+35.29%	1.00	6766.5	+6.15%	1.00
TinyDTLS	2747.5	1901.5	+44.49%	1.00	2052.5	+33.86%	1.00
Average	-	-	+46.52%	-	-	+30.80%	-

both AFLNET and NYX-NET. Initially, at the start of each experiment, all three fuzzers covered a similar number of code branches. However, over time, $\mathcal{E}$FUZZ substantially covered more code than AFLNET and NYX-NET. Even after 24 hours, $\mathcal{E}$FUZZ still had the potential to discover new code, whereas, in most cases, the code coverage for AFLNET and NYX-NET tended to plateau quickly.

Table 4 shows the final code coverage of $\mathcal{E}$FUZZ and two baselines. To quantify the improvement of $\mathcal{E}$FUZZ over baselines, we report the number of branches covered by $\mathcal{E}$FUZZ, AFLNET and NYX-NET (*Coverage*), respectively, the percentage improvement of $\mathcal{E}$FUZZ (*Improv*), and the probability that a random campaign of $\mathcal{E}$FUZZ outperforms a random campaign of baselines ($\hat{A}_{12}$). For all subjects, $\mathcal{E}$FUZZ covers more code than both baselines. Specifically, $\mathcal{E}$FUZZ averagely covers 46.52% more code than AFLNET with a range from 8.39% to 100.69%. When compared to NYX-NET, $\mathcal{E}$FUZZ covers 30.80% more code on average from 3.54% to 101.82%. The Vargha-Delaney [28] effect size $\hat{A}_{12} \geq 0.70$ demonstrates a substantial improvement of $\mathcal{E}$FUZZ over both baselines in terms of code coverage.

To investigate the correlation between improved code coverage and program environment fuzzing, we conducted a comprehensive analysis of the additional code covered by $\mathcal{E}$FUZZ, focusing on the subject DCMTK. DCMTK is a widely-used implementation of the DICOM (Digital Imaging and Communication in Medicine) protocol. While fuzzing DCMTK using $\mathcal{E}$FUZZ, we observed multiple environment sources that undergo mutation. These included the configuration file, the database responsible for storing patient records, various patient cases, and network sockets utilized for hospital communication. Among the 5819 additionally covered branches, 69% of them demonstrated direct connections to environmental mutations, such as parsing and changing the configuration settings and adding entries to the database. Therefore, full environment fuzzing significantly contributes to increased code coverage.

> $\mathcal{E}$FUZZ covers 46.52% and 30.80% more code than AFLNET and NYX-NET, respectively, with most additional code coverage resulting from program environment fuzzing.

Comparing Results on Fuzzing Throughput. The experimental results on fuzzing throughput are shown in Table 5. For each fuzzer, the corresponding fuzzing throughput is shown in the respective columns. In the *Speedup* columns, we present how much faster $\mathcal{E}$FUZZ executes compared to AFLNET and NYX-NET, respectively. $\mathcal{E}$FUZZ achieves a fuzzing throughput ranging from 101.7 to 1320.9 executions per second. The fuzzing throughput of AFLNET is from 3.1 to 38.6 executions per second, and $\mathcal{E}$FUZZ executes 53.85 × faster than AFLNET on average. When compared to NYX-NET, $\mathcal{E}$FUZZ executes faster than NYX-NET on some subjects (e.g., 1.78× faster on Live555) but slower on others. These results are expected as AFLNET always replays each input sequence from the root, while both $\mathcal{E}$FUZZ and NYX-NET avoid replaying repetitive input sequences by faithful replay and state snapshots, respectively. In addition, compared to NYX-NET, $\mathcal{E}$FUZZ introduces some time overhead on certain subjects (e.g., DCMTK) to explore more behaviors. However, this overhead is justified by the evident improvement in bug finding and code coverage. Overall, $\mathcal{E}$FUZZ still maintains a robust throughput.

Table 5: Fuzzing throughput (execs/s) in 10 runs of 24 hours achieved by $\mathcal{E}$FUZZ compared to AFLNET and NYX-NET.

Subject	$\mathcal{E}$FUZZ	Compare with AFLNET		Compare with NYX-NET	
		AFLNET	Speedup	NYX-NET	Speedup
DCMTK	101.7	22.3	4.57×	815.4	0.12×
DNSmasq	393.0	22.6	17.40×	1126.8	0.35×
Exim	713.4	5.1	139.06×	514.5	1.39×
Kamailio	121.9	5.2	23.62×	234.9	0.52×
Live555	237.4	16.8	14.13×	133.1	1.78×
OpenSSH	1320.9	38.6	34.19×	1031.1	1.28×
OpenSSL	124.5	32.7	3.80×	227.4	0.55×
ProFTPD	293.4	7.2	40.91×	333.5	0.88×
Pure-FTPd	528.9	9.8	54.08×	596.0	0.89×
TinyDTLS	640.9	3.1	206.74×	1354.0	0.47×
Average	-	-	53.85×	-	0.82×

> $\mathcal{E}$FUZZ maintains a robust fuzzing throughput while enhancing the capability of bug finding and code coverage.

7.4 Ablation Studies (RQ.3)

Impact of Algorithm Components. $\mathcal{E}$FUZZ employs two strategies to enhance the search efficiency of the program environment: behavior divergence handling based on the relaxed replay, and feedback guidance. To evaluate the impact of each strategy on the improvement of the code coverage, we conducted an ablation study. For this purpose, we developed two ablation tools:

- EF1: based on $\mathcal{E}$FUZZ, without behavior divergence handling,
- EF2: based on $\mathcal{E}$FUZZ, without fuzzing feedback.

We compare the average code coverage achieved by $\mathcal{E}$FUZZ with that of EF1 and EF2 across 10 runs of 24 hours in each subject, and report the percentage improvements.

Table 6 shows the results of the percentage improvements in terms of average code coverage. Overall, across all subjects, both strategies contributed to the increase in code coverage, with none exhibiting a negative impact. Compared to EF1 without behavior divergence handling, $\mathcal{E}$FUZZ resulted in an average increase of 30.59% in code coverage. Notably, in DCMTK, TinyDTLS and MC, $\mathcal{E}$FUZZ exhibited code coverage improvements exceeding 60%. Compared to EF2 without fuzzing feedback, $\mathcal{E}$FUZZ increased the code coverage

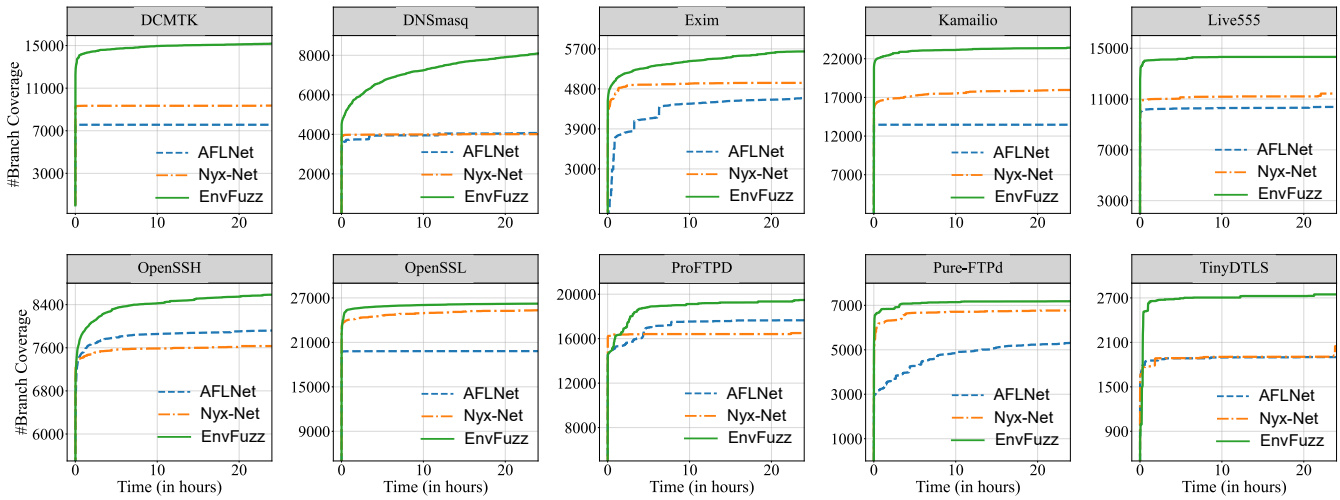

Figure 5: Code covered over time by AFLNet, Nyx-Net and $\mathcal{E}$fuzz across 10 runs of 24 hours on ProFuzzBench subjects.

Table 6: Improvement of code coverage achieved by $\mathcal{E}$fuzz in comparison to ablation tools EF1 and EF2. The results show that the impact of behavior divergence handling and fuzzing feedback is significant.

Subject	*vs.* EF1	*vs.* EF2	Subject	*vs.* EF1	*vs.* EF2
DCMTK	+60.83%	+22.52%	gedit	+22.14%	+8.17%
DNSmasq	+39.28%	+27.79%	Calculator	+27.12%	+6.61%
Exim	+12.24%	+9.66%	Monitor	+14.24%	+4.44%
Kamailio	+28.89%	+10.52%	Glxgears	+12.01%	+2.39%
Live555	+30.26%	+14.61%	MC	+68.60%	+13.46%
OpenSSH	+10.92%	+3.99%	nano	+20.48%	+8.75%
OpenSSL	+12.98%	+8.06%	Vim	+12.50%	+20.47%
ProFTPD	+26.81%	+9.21%	Wireshark	+17.90%	+8.17%
Pure-FTPd	+46.21%	+6.75%	Xcalc	+27.66%	+5.55%
TinyDTLS	+98.57%	+8.59%	Xpdf	+22.19%	+7.79%
Average				+30.59%	+10.38%

by 2.39% to 27.79%, with an average increase of 10.38%. Furthermore, comparing $\mathcal{E}$fuzz with both tools across all subjects, $\hat{A}_{12}$=1, which indicates that $\mathcal{E}$fuzz significantly outperforms EF1 and EF2. These results demonstrate the importance of $\mathcal{E}$fuzz's divergence handling and the effectiveness of fuzzing feedback in guiding the search.

We further measured the fuzzing throughput of $\mathcal{E}$fuzz, EF1, and EF2 across each subject. On average, $\mathcal{E}$fuzz achieves a fuzzing throughput of 447.6 executions per second, while EF2 achieves a similar throughput of 454.9 executions per second. However, EF1 executes faster than $\mathcal{E}$fuzz with a fuzzing throughput of 698.3 executions per second. This higher throughput is due to $\mathcal{E}$fuzz's strategy of handling behavior divergence, which explores longer traces, trading raw throughput for better coverage.

Divergence handling and feedback guidance enable $\mathcal{E}$fuzz to increase code coverage by 30.59% and 10.38%, respectively. The contribution of each strategy to enhancing code coverage is significant.

Analysis of Relaxed Replay. To further analyze the impact of relaxed replay for divergence handling, we examined the following additional questions:

- How often do executions resort to relaxed replay (#Freq.)?
- How many system calls in tree branches use relaxed replay (#RelaxSysCs *vs* #TotalSysCs)?
- How early after a branch does relaxed replay start (#StartPoint)?

For this purpose, we collect the statistical data from 20 subjects over 24-hour runs and report them in Table 7. On average, 84.48% of all executions across the 20 subjects have to resort to relaxed replay. The total number of system calls executed in each tree branch (calculated from forking points) is 187.9, with 108.1 of those system calls using relaxed replay. In addition, the starting points of the relaxed replay vary among different subjects, ranging from 5.46% of the tree branch on TinyDTLS to 89.87% on OpenSSH. On average, the relaxed replay starts to resort to relaxed replay around halfway (49.97%). These results demonstrate that the relaxed replay for divergence handling is necessary and commonly used by $\mathcal{E}$fuzz.

7.5 Discussion

Manual Effort. The manual effort needed for using $\mathcal{E}$fuzz is minimal. The only manual involvement is the user inputs necessary for testing GUI applications in the recording phase. For example, when testing the calculator, the user needs to open the application, execute a simple addition operation, and then close it. After recording, the rest of the fuzzing workflow is fully automatic. To collect code coverage feedback, $\mathcal{E}$fuzz can directly instrument the binaries of the program under test, eliminating the need to recompile from source code. Similarly, the system-call interception infrastructure (for record and replay) is designed to work with binary code.

Limitations. In this paper, we leverage greybox fuzzing over complex program environments. We have demonstrated that the approach of $\mathcal{E}$fuzz is effective in exposing previously unknown bugs and enhancing code coverage. Like other fuzzers, the efficacy of $\mathcal{E}$fuzz depends on the quality of the initial seed(s), and $\mathcal{E}$fuzz is not

Table 7: Statistical analysis of relaxed replay proposed by $\mathcal{E}$FUZZ, including the frequency of the executions resorting to relaxed replay (#Freq.), the total number of system calls executed in each tree branch (#TotalSysCs), the number of system calls resorting to relaxed replay in each tree branch (#RelaxSysCs), and the point at which a tree branch starts to resort to relaxed replay (#StartPoint).

Subject	#Freq.	#TotalSysCs	#RelaxSysCs	#StartPoint
DCMTK	93.67%	130.0	116.7	10.27%
DNSmasq	34.52%	99.1	11.1	88.82%
Exim	98.78%	90.8	23.4	74.18%
Kamailio	73.85%	141.2	90.5	35.93%
Live555	45.12%	375.1	130.2	65.30%
OpenSSH	87.66%	105.3	10.7	89.87%
OpenSSL	91.02%	51.3	11.0	78.57%
ProFTPD	95.01%	172.1	24.8	85.61%
Pure-FTPd	60.08%	114.9	14.5	87.41%
TinyDTLS	78.58%	312.6	295.5	5.46%
gedit	99.89%	397.3	332.1	16.42%
Calculator	99.65%	152.1	80.7	46.98%
Monitor	94.68%	114.5	69.6	39.21%
Glxgears	94.31%	92.6	35.8	61.37%
MC	89.98%	278.1	127.6	54.11%
nano	98.99%	164.6	89.0	45.93%
Vim	93.69%	386.2	307.7	20.32%
Wireshark	81.43%	198.4	157.5	20.60%
Xcalc	89.37%	147.5	51.7	50.78%
Xpdf	89.30%	234.1	182.0	22.26%
Average	84.48%	187.9	108.1	49.97%

guaranteed to cover the entire search space. Limited seed recordings (e.g., open and immediately close a GUI) generally result in a limited exploration compared to diverse recordings (e.g., exercising different GUI elements). However, the dependence on quality seeds is an inherent limitation of fuzzing in general, in contrast to model checking and other verification techniques that attempt to systematically explore the entire search space. While $\mathcal{E}$FUZZ can fuzz a broad range of programs, its scope is limited to Linux user-mode environments. This limitation stems from our underlying environmental record and replay infrastructure. Despite this, and compared to existing fuzzers, $\mathcal{E}$FUZZ still maintains its generality, and can fuzz even challenging subjects such as network protocols and GUI applications.

Relaxed replay assumes that I/O system calls can be mutated and reordered arbitrarily. This is a straightforward generalization of what existing fuzzers already assume. For example, AFL implicitly assumes the input file can be mutated arbitrarily, while AFLNET assumes messages can be reordered. However, these assumptions may not always hold for some edge cases. Special files (e.g., `/proc/*` and `/dev/zero`) and self-pipes are not mutable. Fortunately, such examples are rare and can be avoided using a predefined special-case list. As such, no false positives were detected during our evaluation. By design, $\mathcal{E}$FUZZ does not use modeling, allowing it to fuzz programs without any manual effort or prior knowledge. In addition, $\mathcal{E}$FUZZ supports fuzzing "new" syscalls not present in the original recording. During relaxed replay, $\mathcal{E}$FUZZ provides inputs to the syscall based on file descriptors, regardless of the specific syscall number. However, due to behavior divergence, if the program invokes a system call to access inputs from a file descriptor that is not originally recorded, $\mathcal{E}$FUZZ will fail the system call to maintain the plausibility of the replay.

8 RELATED WORK

Environment Capture. Environment handling poses a critical challenge in the realm of model checking and symbolic execution, where achieving an accurate analysis of program behaviors requires considering the full surrounding environment. Many existing approaches manually abstract the environment via a model [6, 10, 19, 25, 35], but crafting abstract models is labor-intensive. Some alternatives [12, 32] leverage virtualization to eliminate the need for constructing abstract models. However, the path-explosion problem persists when analyzing an entire software stack [5, 12]; the presence of many program environments further exacerbates the path-explosion problem while finding bugs in software.

Fuzzing Effort. In the area of fuzzing, existing fuzzers often focus on a single input, disregarding other environment sources. The potential solutions for capturing environment effects, are the use of Virtual Machine (VM) fuzzing [34] allowing the target to be fuzzed in the context of an emulated system environment, or overriding `glibc` functions [24]. VM-based fuzzing is a heavyweight solution. Moreover, both approaches cannot hook the full environment, which misses environmental interactions with external servers, hardware devices, and human users. In contrast, our approach is lightweight yet robust, effectively handling the full environment.

Stateful Fuzzing. Many programs are stateful, processing inputs based on their internal states. While fuzzing stateful programs, relying solely on code coverage is insufficient in guiding fuzzers to explore complex state machines and reach deep states [3, 4, 23, 31, 33]. Identifying program states poses a significant challenge, and several works propose diverse state representation schemes. IJON [3] uses human code annotations to annotate states, and AFLNET [31] manually extracts response code based on network protocols as states (e.g., 404 for `http`). STATEAFL [26] hashes in-memory variables as states, while SGFUZZ [4] and NSFUZZ [33] utilize enum variables as states with manual filtering. However, these approaches involve much manual effort or employ specific heuristics such as the emphasis on enum variables in SGFUZZ.

Snapshot Fuzzing. When fuzzing stateful systems, achieving a deep exploration of program states often requires a lengthy sequence of messages. For instance, AFLNET [31] opts for replaying each message sequence from initial states, somewhat impeding its fuzzing speed. To address this limitation, SNAPFUZZ [2] employs an in-memory filesystem to efficiently reset to specific interesting states, overcoming the impediment faced by AFLNET. In a similar vein, NYX-NET [34] introduces a hypervisor-based technique to dump program states at points of interest, including all memory contents. Our algorithm eliminates the need for snapshots or hypervisors, and dynamically *reconstructs* states on-demand through replay. Our algorithm has similarities with `fork`-based fuzzers such as AFL [37] and AFL++ [17]. Rather than employing a global fork server at program entry, we implement a mini-fork server at each program input, avoiding replaying system call sequence prefixes.

Record and Replay. Record and replay have been widely used in assisting program analysis [13, 15, 29, 38]. These approaches have targeted different software and hardware, including virtual machines [13, 15], user-space programs [22, 29] and hardware [30, 38]. Among them, `rr` [29] is a well-known debugger for its ease of use and low adoption cost. Its design principle is to record and replay unmodified user-space applications (binaries) with stock Linux kernels, with a fully user-space implementation running without special privileges, and without using pervasive code instrumentation. The `rr` debugger is primarily designed to help with difficult-to-reproduce bugs that depend on nondeterministic elements of the environment. Our approach has some similarities with the `rr` debugger, but we re-purpose `rr` debug-style replay for bug discovery by employing relaxed replay.

9 CONCLUSION

In this paper, we have proposed a methodology, tool, and evaluation to handle complex program environments. Our $\mathcal{E}$FUZZ tool avoids environment modeling by recording program executions and selectively mutating (in the style of greybox fuzzing) the recorded executions during replay to capture the effect of different environments. Evaluation of $\mathcal{E}$FUZZ found 33 previously unknown bugs, out of which 24 were confirmed by developers. The applications tested include well-known GUI applications and protocol implementations. $\mathcal{E}$FUZZ presents a general approach for handling software environments, which is different from (a) the practitioners' approach of procuring sample environments for testing code on them one by one, or (b) the current established research on environment modeling. We do not model environments and we do not procure environments. Instead $\mathcal{E}$FUZZ is an automated framework for *implicitly* navigating the space of program environments via mutational fuzzing.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their suggestions. We also thank Van-Thuan Pham for his comments on the draft. This research is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing NRF-NCR25-Fuzz-0001). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore.

REFERENCES

[1] AFLplusplus. Afl++. https://github.com/AFLplusplus/AFLplusplus.
[2] Anastasios Andronidis and Cristian Cadar. Snapfuzz: high-throughput fuzzing of network applications. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 340–351, New York, 2022. ACM.
[3] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, pages 1597–1612. IEEE, 2020.
[4] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272. USENIX Association, 2022.
[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.
[6] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review*, 40(4):73–85, 2006.
[7] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Software*, 38(3):79–86, 2020.
[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.
[9] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1621–1633. ACM, 2022.
[10] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.
[11] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security*, 12(2):1–38, 2008.
[12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–278. ACM, 2011.
[13] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, pages 1–11. ACM, 2015.
[14] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, pages 151–163. ACM, 2020.
[15] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.
[16] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 337–350. ACM, 2021.
[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies*. USENIX Association, 2020.
[18] Xiang Gao, Gregory J. Duck, and Abhik Roychoudhury. Scalable fuzzing of program binaries with e9afl. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1247–1251. ACM, 2021.
[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223. ACM, 2005.
[20] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium*, 2021.
[21] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138. ACM, 2018.
[22] Christopher Lidbury and Alastair F Donaldson. Sparse record and replay with controlled scheduling. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 576–593. ACM, 2019.
[23] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium*. ISOC, 2024.
[24] Zahra Mirzamomen and Marcel Böhme. Finding bug-inducing program environments. *arXiv preprint arXiv:2304.10044*, 2023.
[25] Madanlal Musuvathi, David YW Park, Andy Chou, Dawson R Engler, and David L Dill. Cmc: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, 36(SI):75–88, 2002.
[26] Roberto Natella. Stateafl: Greybox fuzzing for stateful network servers. *Empirical Software Engineering*, 27(191), 2022.
[27] Roberto Natella and Van-Thuan Pham. Profuzzbench: A benchmark for stateful protocol fuzzing. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, pages 662–665. ACM, 2021.
[28] Geoffrey Neumann, Mark Harman, and Simon Poulding. Transformed vargha-delaney effect size. In *Proceedings of Search-Based Software Engineering: 7th International Symposium*, pages 318–324. Springer, 2015.
[29] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *Proceedings of the 2017 USENIX Annual Technical Conference*, pages 377–389. USENIX Association, 2017.
[30] Heejin Park and Felix Xiaozhu Lin. Gpureplay: a 50-kb gpu stack for client ml. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–170. ACM, 2022.

[31] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *Proceedings of the 13th IEEE International Conference on Software Testing, Verification and Validation: Testing Tools Track*, pages 460–465. IEEE, 2020.

[32] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with {SymCC}: Don't interpret, compile! In *Proceedings of the 29th USENIX Security Symposium*, pages 181–198. Usenix Association, 2020.

[33] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. Nsfuzz: Towards efficient and state-aware network service fuzzing. *ACM Transactions on Software Engineering and Methodology*, 32(160):1–26, 2023.

[34] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: network fuzzing with incremental snapshots. In *Proceedings of the 17th European Conference on Computer Systems*, pages 166–180. ACM, 2022.

[35] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.

[36] O. Tkachuk, M.B. Dwyer, and C.S. Pasareanu. Automated environment generation for software model checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 116–127. IEEE, 2003.

[37] Michał Zalewski. AFL. https://lcamtuf.coredump.cx/afl/.

[38] Gefei Zuo, Jiacheng Ma, Andrew Quinn, and Baris Kasikci. Vidi: Record replay for reconfigurable hardware. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 806–820. ACM, 2023.