G-Fuzz: A Directed Fuzzing Framework for gVisor

Yuwei Li, Yuan Chen, Shouling Ji, Xuhong Zhang, Guanglu Yan, Alex X. Liu,
Chunming Wu, Zulie Pan, Peng Lin

Abstract—gVisor is a Google-published application-level kernel for containers. As gVisor is lightweight and has sound isolation, it has been widely used in many IT enterprises [1]–[3]. When a new vulnerability of the upstream gVisor is found, it is important for the downstream developers to test the corresponding code to maintain the security. To achieve this aim, directed fuzzing is promising. Nevertheless, there are many challenges in applying existing directed fuzzing methods for gVisor. The core reason is that existing directed fuzzers are mainly for general C/C++ applications, while gVisor is an OS kernel written in the Go language. To address the above challenges, we propose G-Fuzz, a directed fuzzing framework for gVisor. There are three core methods in G-Fuzz, including lightweight and fine-grained distance calculation, target related syscall inference and utilization, and exploration and exploitation dynamic switch. Note that the methods of G-Fuzz are general and can be transferred to other OS kernels. We conduct extensive experiments to evaluate the performance of G-Fuzz. Compared to Syzkaller, the state-of-the-art kernel fuzzer, G-Fuzz outperforms it significantly. Furthermore, we have rigorously evaluated the importance for each core method of G-Fuzz. G-Fuzz has been deployed in industry and has detected multiple serious vulnerabilities.

Index Terms—gVisor, OS Kernel, Directed Fuzzing, Vulnerability Detection

---◆---

1 INTRODUCTION

gVisor [4] is an application kernel that aims to provide secure isolation between the host kernel and the applications running inside the containers. Compared to the virtual machine, gVisor is more lightweight and can provide a similar isolation level. Thus, it has been adopted in many IT companies [1]–[3].

The vulnerabilities of gVisor may severely impact the security and stability of the production environments. Those IT companies usually implement a customized gVisor based on the specific application scenario, and they may face many security issues. How to test whether the modified code of the customized gVisor introduces new vulnerabilities? When a vulnerability of gVisor has been patched, whether the patch can fix the vulnerability? To address the above

- S. Ji and C. Wu are with the corresponding authors. Y. Li and Y. Chen are the co-first authors.
- Y. Li is with the College of Electronic Engineering, National University of Defense Technology, Hefei, Anhui, 230000, China. Part of this work was down when she was a Ph.D student at the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang, 310027, China. Email: liyuwei@nudt.edu.cn
- S. Ji is with the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang, 310027, China. Email: sji@zju.edu.cn
- X. Zhang is with the School of Software Technology, Zhejiang University, Hangzhou, Zhejiang, 310027, China. Email: zhangxuhong@zju.edu.cn
- G. Yan and A. Liu are with Ant Group, Hangzhou, Zhejiang, 310063, China. Email: flankreader@gmail.com, alexliu@antgroup.com
- Y. Chen and C. Wu are with College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang, 310027, China. Email: chenyuan@zju.edu.cn, wuchunming@zju.edu.cn
- Z. Pan is with the College of Electronic Engineering, National University of Defense Technology, Hefei, Anhui, 230000, China. Email: panzulie17@nudt.edu.cn
- P. Lin is with Chinese Aeronautical Establishment and AVIC Artificial Intelligence Research Institute, Beijing, 100029, China. Email: 13940001294@163.com

questions, directed fuzzing is one promising and practical solution. Directed fuzzing aims at generating the test cases that can trigger the specified target code. The current directed fuzzing techniques can be mainly categorized into two classes: symbolic execution based whitebox directed fuzzing [5], [6] and directed greybox fuzzing (DGF) [7], [8]. As symbolic execution has limitations such as path explosion and complex constraints, it is hard to apply symbolic execution based directed fuzzing on large and complex software like kernels. For OS kernels, compared to symbolic execution based methods, DGF has better scalability and is more feasible. Nevertheless, there are still many challenges when applying existing DGF techniques on gVisor as follows.

High Time Overhead in Distance Calculation. DGF usually leverages distance information as the guidance for testing the target. Specifically, DGF selects the closer inputs as seeds to generate new inputs. For each code (e.g., function/basic block) of the test program, DGF calculates its static distance to the target function/basic block before fuzzing. Then, during the fuzzing process, the executed inputs that cover the code closer to the target will be selected as the seeds to further generate new inputs. However, the time overhead of the current DGF in calculating static distance is hardly affordable for large software or systems like OS kernels. For instance, when leveraging the state-of-the-art DGF method, AFLGo [7], to calculate the static distance for gVisor, the average time it takes is more than 16 hours. Therefore, the time burden brought by the current static distance methods severely impedes the efficiency of directed fuzzing.

The Limitations of the Distance Information. Even with the distance information, there are still many challenges. First, the distance information may not be precise.

For instance, when constructing a program's function Call Graph (CG), AFLGo does not consider the indirect calls. As a result, the CG may lose some critical edges, which further affects the precision of the distance information. In addition, both AFLGo and Hawkeye [8], the state-of-the-art directed greybox fuzzers, approximate basic block level distance by multiplying function level distance with a constant (e.g., 10 in AFLGo paper [7]), which may cause many biases. Second, although the distance information can provide guidance in triggering the target, it still has disadvantages. The paths that are closer to the target may not necessarily be easier to get there, as distance is spatial information and may not provide an equivalent measure of the difficulty in reaching the target. Moreover, in patch testing, a typical application scenario of directed fuzzing, there may be multiple paths leading to the patch target. Therefore, only focusing on the shortest path might not be able to comprehensively test the patch code. Thus, to improve DGF, it is necessary to improve the precision of the distance calculation and make reasonable use of the distance information.

Difference of Inputs. Existing DGF mainly focuses on testing the general user-space applications that take files as inputs. Although gVisor runs in the user mode, in essence, it is an OS kernel that takes syscall[①] sequences as inputs. Compared to files, the structure and semantic requirements for syscall sequences are more strict. When testing kernels, it is necessary to provide meaningful syscall sequences as inputs. Otherwise, only the shallow code of the kernel can be triggered. gVisor implements more than 200 Linux syscalls, and each syscall may have several parameters. As a result, the whole input space for fuzzing gVisor is extremely vast. However, for directed fuzzing, the target is triggered by a limited syscalls. Thus, identifying the target related syscalls to reduce the input space needed to be explored is crucial to directed fuzzing for kernels.

To address the above challenges, we propose G-Fuzz, a directed fuzzing framework for gVisor, including three principal methods.

Lightweight and Fine-grained Distance Calculation. To solve the problem of the high time overhead and low precision in distance calculation, we propose a *lightweight and fine-grained distance calculation* method. First, we perform *reachability analysis* to find the paths that can lead to the target. We then calculate the distance only for the code on these reachable paths instead of all code. Second, we utilize the *Breadth First Search (BFS)*, a less complex algorithm than the Dijkstra algorithm [9] that used by existing DGF [7], [8], to calculate the distance between two nodes. To solve the false negatives, we use type analysis to identify the indirect calls. Then, we construct an inter-procedural CFG to calculate basic block level distance, which is more fine-grained than existing DGF's distance.

Target Related Syscall Inference and Utilization. To reduce the input space that needs to be explored, we propose a *target related syscall inference and utilization* method. Before directed fuzzing, G-Fuzz automatically infers which syscalls are related to the specified target. Specifically, we propose eight inference rules based on static analysis and the expert knowledge of the gVisor code. Then, based on the inference

results, G-Fuzz utilizes the inferred syscalls in the mutation operations to generate new inputs during the fuzzing process. To make efficient use of the inferred syscalls, we adjust the selection probability of each syscall according to the dynamic information of the fuzzing process. In addition, considering the dependencies of different syscalls, we adjust the order of each syscall in the mutation process to improve the semantic correctness of the generated inputs.

Exploration and Exploitation Dynamic Switch. To make reasonable use of the distance information, we propose the *exploration and exploitation dynamic switch* method. There are two modes of G-Fuzz in the fuzzing process: *exploration* and *exploitation*. In the *exploration* mode, G-Fuzz acts like a "coverage-based" fuzzer, aiming at covering as many paths as possible, which can increase the diversity of the seeds and mitigate false negatives (e.g., missing CG edges). In the *exploitation* mode, G-Fuzz pays more attention to the target by selecting the closer seeds, which aims at accelerating triggering the target. Exploring more paths may increase the probability of triggering the target, but it is not efficient. Only focusing on testing the closer paths may make the fuzzing fall into local optimal. Therefore, it is necessary to make a trade-off and an adaptive adjustment. G-Fuzz adopts a dynamic strategy to adjust the mode selection according to the feedback from the current fuzzing state. During the fuzzing process, if one mode does not make any progress over a time threshold, G-Fuzz will switch to the other mode adaptively.

To evaluate G-Fuzz, we test its performance in three typical application scenarios of directed fuzzing, including general target testing, patch testing and bug reproduction. We compare G-Fuzz to Syzkaller and Syz-Go. Syzkaller is a state-of-the-art kernel fuzzer, and Syz-Go is implemented by us by applying the advanced directed fuzzing methods of AFLGo on Syzkaller. The experimental results demonstrate that G-Fuzz achieves more efficient and stable performance than both Syzkaller and Syz-Go. Out of the evaluated 59 typical targets, G-Fuzz outperforms Syzkaller on 58 ones, and outperforms Syz-Go on 53 ones. G-Fuzz achieves at least twice speed in trigger the targets than Syzkaller and Syz-Go on 32 targets. On 5 targets, only G-Fuzz can successfully trigger while Syzkaller cannot trigger. On 2 targets, only G-Fuzz can successfully trigger while Syz-Go cannot trigger. We also evaluate the effectiveness of each core method of G-Fuzz separately in § 6, and the results show that all methods make important contribution to the outstanding performance of G-Fuzz. Furthermore, we have deployed G-Fuzz in Ant Group, a world-leading IT company, and G-Fuzz has detected multiple real-world vulnerabilities of its customized gVisor.

The contributions of our paper are summarized as follows.

- **Novel and General Methods.** We propose G-Fuzz, a directed fuzzing framework for gVisor, with three novel methods: *lightweight and fine-grained distance calculation*, *target related syscall inference and utilization* and *exploration and exploitation dynamic switch*. The methods of G-Fuzz are general and scalable, which can be easily extended to testing more OS kernels. Based on the proposed methods of G-Fuzz, we also implement a

---

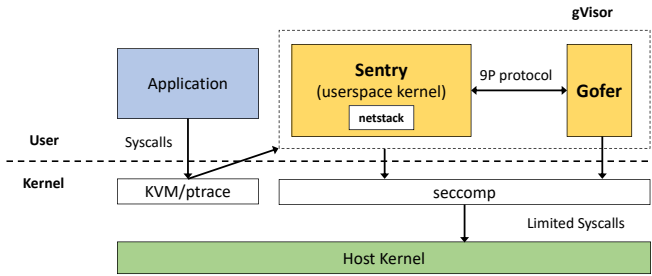[①]We use syscall to indicate system call in this paper.

Fig. 1: The framework of gVisor.

prototype called G-Fuzz-Linux, for directed fuzzing Linux kernels.

- **Significant Performance.** We conduct extensive experiments to evaluate the performance of G-Fuzz. The experimental results demonstrate that G-Fuzz outperforms the state-of-the-art kernel fuzzer significantly.
- **Real-world Impacts.** We have applied G-Fuzz in practice and leverage it to discover multiple real-world vulnerabilities of the downstream gVisor in industry. To facilitate further research on OS kernel directed fuzzing, we have open-sourced both G-Fuzz [10] and G-Fuzz-Linux [11].

2 BACKGROUND

In this section, we first give a brief introduction about gVisor. Then, we introduce directed fuzzing techniques.

2.1 gVisor

gVisor is an application kernel, which is mainly used in virtual environments such as containers. It implements the isolation by intercepting the syscalls requested from the applications in containers, acting as a guest kernel. gVisor supports most Linux syscalls and runs as a standard, unprivileged process in user-space. Specifically, gVisor supports 260 syscalls for the AMD64 arch in version `release-20210125.0`. The yellow color part of Fig. 1 presents the framework of gVisor. It consists of two primary components: Sentry and Gofer, which run as two processes. Sentry is the core of gVisor, which is responsible for processing the syscalls from the user-space applications. gVisor provides two modes: KVM and ptrace, to redirect the syscalls to Sentry. To provide extra security protection, gVisor adopts the `seccomp (secure computing mode)` mechanism to restrict the available syscalls to the host kernel. For instance, the Sentry process has no access to file-related syscalls. Therefore, when Sentry needs to read or write files on the host file system, it will communicate with the Gofer process using the 9P protocol. Gofer mediates all these host file-system accesses, providing an additional level of isolation. gVisor also implements a user-space network stack (i.e., `netstack`), which can process most network-related tasks.

2.2 Directed Fuzzing

Given a target site of a program, the goal of directed fuzzing is to generate the inputs that can trigger it. Compared to coverage-based fuzzing, directed fuzzing is faster in detecting vulnerabilities that locate in specified sites. The original directed fuzzing methods are symbolic execution based [5], [12]–[14], which cast the reachability problem as the iterative constraint satisfaction problem [7]. Nevertheless, due to the heavyweight program analysis and the difficulty of constraint solving, these methods suffer from the issues of lousy scalability.

To solve these issues, DGF techniques [7], [8] are proposed, which leverage the distance information to guide the fuzzing in generating inputs that can reach the target. Before the fuzzing, DGF calculates a static distance for each component (e.g., basic block or function) of a program to the target. During the fuzzing process, the inputs that are closer to the target are selected as seeds and given more mutation times. The distance of an input is calculated based on the static distance of its covered paths. Compared to symbolic execution based directed fuzzing methods, DGF is more practical and has better performance. For instance, AFLGo, a state-of-the-art DGF, spends less than 20 minutes triggering the *heartbleed* vulnerability, while KATCH [5], a state-of-the-art symbolic execution based fuzzer, cannot trigger this vulnerability within 24 hours [7]. Therefore, in this paper, we choose to leverage the DGF method for testing gVisor. There are many works to improve the efficiency of DGF [15]–[19]. Nevertheless, the above directed fuzzing work is still not appropriate for testing OS kernels. As we discuss in §1, there are many challenges in applying existing DGF techniques on kernels. That is what we aim to solve in this paper.

3 DESIGN OF G-FUZZ

In this section, we first present the overview of G-Fuzz. Then we describe each core method of G-Fuzz in detail.

3.1 Overview

Fig. 2 illustrates the overview of G-Fuzz, which consists of two main components: static analysis and directed fuzzing. Given a target in gVisor, the static analysis component aims to extract the information for directed fuzzing, including static distance, target related syscalls, etc. Then, the directed fuzzing component leverages the extracted information to steer the fuzzer in generating inputs that can trigger the target.

Static Analysis. There are four main steps involved in the static analysis component of G-Fuzz. First, G-Fuzz constructs the CG and CFG of gVisor. Second, based on the CG and CFG, G-Fuzz performs *reachability analysis* for the given target to find the paths in gVisor which can lead to the target. We name the code on these paths as the *reachable set* of the target. Third, G-Fuzz calculates the static distance for each basic block in the *reachable set*. Finally, G-Fuzz determines which syscalls are related to the target based on the automatic inference rules, which are designed based on static analysis and expert knowledge. In addition, G-Fuzz leverages the static distance information to measure the relevance between each inferred syscall with the target.

Directed Fuzzing. After static analysis, G-Fuzz moves on to directed fuzzing. The initial seeds are constructed based on the inferred target related syscalls. Next, G-Fuzz
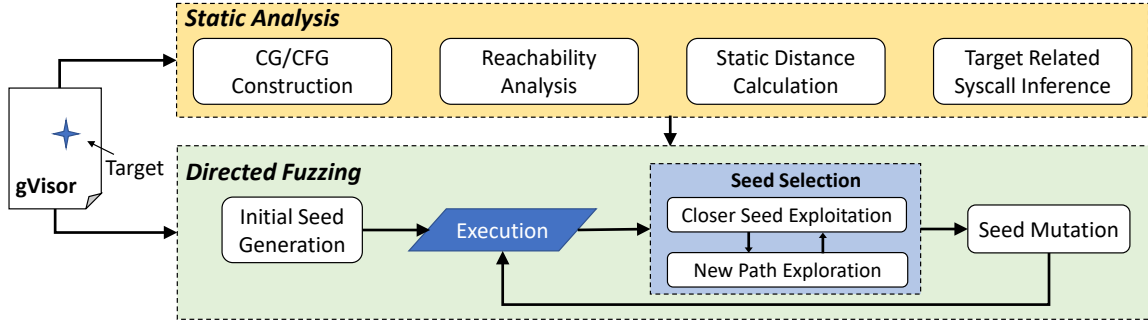
Fig. 2: The overview of G-Fuzz.

selects the inputs with good performance as seeds throughout the fuzzing process. There are two strategies for selecting inputs. One is selecting inputs that find new paths (i.e., *exploration* strategy), and the other is selecting inputs closer to the target (i.e., *exploitation* strategy). To make reasonable use of the distance information and alleviate its limitations, we propose an *exploration and exploitation dynamic switch* method to seek a balance. When the fuzzer gets "stuck" by using a seed selection strategy over a time threshold, G-Fuzz will switch to the other strategy adaptively.

As a mutation-based fuzzer, G-Fuzz mutates the seeds to generate new inputs. Considering that gVisor is an OS kernel, whose inputs are syscall sequences, we propose the following methods in G-Fuzz to generate high-quality inputs for directed fuzzing. First, G-Fuzz leverages the inferred syscalls in the mutation process to generate inputs that tend to trigger the target. Second, as there may exist false positives in the inferred syscalls, G-Fuzz dynamically adjusts the selection probability of each syscall according to its effectiveness in directed fuzzing. Third, G-Fuzz does not insert the syscalls randomly but determines their orders based on the dependencies of different syscalls. Based on the above methods, G-Fuzz can generate semantically correct inputs that have close relevance to the target.

Below, we will give a detailed description of the methods in G-Fuzz.

3.2 Lightweight and Fine-grained Distance Calculation

When applying the existing DGF techniques in OS kernels like gVisor, there are two main issues with their static distance calculation methods. One is that the time overhead of distance calculation is unacceptably high. The other is that the distance information is coarse-grained and imprecise. Next, we first provide a depth analysis on the two challenges in detail and then propose our solutions.

High Time Overhead. The time overhead of existing DGF in calculating static distance is hardly affordable for large systems like OS kernels. There are two main state-of-the-art directed greybox fuzzers: AFLGo and Hawkeye. As only AFLGo is open-source, we can only test AFLGo and analyze its code. Next, we mainly take AFLGo as the example. For instance, AFLGo spends nearly 2 hours in compiling and instrumenting `cxxfilt` [20]. Compared to `cxxfilt`, gVisor is an OS kernel, which has more code. In our experiments, we find that AFLGo spends more than 16 hours in average in calculating the static distance for gVisor. Note that when testing a new target, the distance

information needs to be re-calculated. To address the high time overhead issue, we conduct an in-depth analysis about the method and code of existing DGF in calculating distance and find the following causes.

First, AFLGo calculates the distance between all code and the target with a traversal method. Nevertheless, only parts of the code can lead to the target. Thus, it is unnecessary to calculate distances for all code. Second, both AFLGo and Hawkeye utilize the Dijkstra algorithm to find the shortest path between two nodes to further calculate their distance. The Dijkstra algorithm is mainly used to find the shortest path for weighted graphs. As both CG and CFG are unweighted graphs, it is unnecessary to use the Dijkstra algorithm. Third, the implementation code of AFLGo has many issues. For instance, each time AFLGo calculates a function level distance, it starts a Python process, incurring unnecessary Python startup and library initialization time cost repeatedly. Moreover, the static distance information is instrumented into the target program. That is, the high time overhead distance calculation is tightly coupled to the compilation process. When testing other targets, in addition to re-calculating the static distance, AFLGo has to re-build the target program, causing much overhead.

Coarse-grained and Imprecise Distance. Despite the time overhead issue, the distance information calculated by the existing DGF methods is inadequate in terms of granularity and precision. First, AFLGo and Hawkeye approximate basic block level distance with function level distance, which is coarse-grained and may cause much bias. Second, when constructing CG, AFLGo does not consider the indirect calls. Thus, the CG may miss some edges, which further affects the precision of the distance information. Hawkeye uses *Andersen*'s pointer analysis method [21] to identify indirect calls. However, pointer analysis is computational expensive [22] and does not guarantee the soundness [23], [24].

Solutions of G-Fuzz. To address the above issues, we propose a lightweight and fine-grained distance calculation method. First, G-Fuzz constructs the CG of gVisor and performs intra-procedural analysis to extract the CFG of each function in gVisor. We leverage *Rapid Type Analysis (RTA)* [25], a type analysis method, and customize `go-callvis` [26] to identify the indirect calls in gVisor. Compared to pointer analysis, type analysis is faster and has fewer false negatives. We find that there exist false negatives (i.e., missing some critical indirect calls) when using pointer analysis for gVisor. Missing critical indirect

```go
func (rfd *replicaFileDescription) Ioctl(           ctx
    context.Context, io usermem.IO,        args arch.
    SyscallArguments) (uintptr, error) {
    ...
    switch cmd := args[1].Uint(); cmd {
        case linux.FIONREAD:
            return 0, rfd.inode.t.ld.inputQueueReadSize(t,
                io, args)
        case linux.TCGETS:
            return rfd.inode.t.ld.getTermios(t, args)
        case linux.TCSETS:
            return rfd.inode.t.ld.setTermios(t, args)
        ...
}
```

Fig. 3: A code snippet related with `ioctl`.

TABLE 1: The information of the syscall variants that G-Fuzz can infer.

Syscall	Variants Count	Example
arch_prctl	3	arch_prctl$ARCH_GET_FS
epoll_ctl	3	epoll_ctl$EPOLL_CTL_MOD
getsockopt	10	getsockopt$inet6_IPV6_IPSEC_POLICY
ioctl	49	ioctl$TCGETS2
prctl	21	prctl$PR_SET_MM_AUXV
semctl	15	semctl$SEM_INFO
setsockopt	16	setsockopt$inet_MCAST_JOIN_GROUP
shmctl	9	shmctl$SHM_LOCK
waitid	1	waitid$P_PIDFD

calls may make CG lose many edges, which will severely impact the guidance of directed fuzzing. Thus, we choose to use type analysis although it may over-approximate some indirect calls. Based on the extracted CG, G-Fuzz performs *reachability analysis* on gVisor for the given target to find out which functions are on the paths to the target. We name these functions as *reachable functions*. In specific, starting from the target function, G-Fuzz adopts a bottom-up method to traverse the callers iteratively to obtain all the *reachable functions*. Then, based on the CG, G-Fuzz connects the CFGs for these *reachable functions* to construct a local inter-procedural CFG. Next, G-Fuzz utilizes the *BFS* algorithm to find the shortest path between each node of the local inter-procedural CFG and the target node. The length of the shortest path is the basic block level distance.

In summary, our proposed method can effectively address the issues of high time overhead and low precision in distance calculation. For the time overhead issue, first, by performing *reachability analysis*, we reduce the overhead caused by the unrelated code. Second, we use the *BFS* algorithm rather than the Dijkstra algorithm to find the shortest paths, which has a lower complexity. More specifically, the complexity of the Dijkstra algorithm is $O(V^2)$, while the complexity of the *BFS* algorithm is $O(V + E)$, where V represents the number of nodes in the graph, and E represents the number of edges in the graph. For the low precision issue, first, we utilize type analysis to identify the indirect calls of CG to reduce the false negatives. Second, based on the *reachability analysis*, we can effectively construct a local inter-procedural CFG to calculate the more precise basic block level distance than existing DGF. In addition, we do not instrument the distance information into the gVisor, but use a file to map each basic block and its static distance, making the method more scalable than existing DGF methods. We conduct experiments to provide detailed evaluation and analysis about the overhead and precision of our distance calculation method. Specifically, for six different gVisor's targets, G-Fuzz spends 97.1 seconds in calculation the static distance for gVisor in average, while AFLGo spends 68346 seconds in average, demonstrating the significant performance of G-Fuzz. More detailed results are presented in §6.1.

3.3 Inference of Target Related Syscalls

In essence, the goal of directed fuzzing gVisor is to generate the syscall sequences that can trigger the specified target. However, generating the inputs which can trigger the target from the whole input space is quite challenging. gVisor implements more than 200 Linux syscalls, and each syscall may have multiple parameters. Moreover, a sequence may contain different numbers of syscalls, and the permutations of their order are various. Thus, the whole input space is extremely vast. To reduce the input space needed to explore and improve the efficiency of directed fuzzing, we propose an automatic method to infer which syscalls are related to the given target before fuzzing. This method is based on static analysis and the expert knowledge of gVisor. It needs to note that the generation process of the inference rules relies on the expert knowledge. But once the inference rules are formed, they can be used automatically. The usage process is as follows: a user just provides the specific location of the target or crash report[①], and the inference rules automatically output its related syscalls. During the next dynamic fuzzing process, the utilization of the inferred syscalls is also automatic. G-Fuzz has incorporated the following eight inference rules and can be further extended to incorporate more rules.

Function Call Chain based Inference. This is a general rule for all the targets of gVisor. First, based on the *reachability analysis* of the given target, we can obtain the *reachable functions* that lead to the target. Some of the reachable functions are syscall handlers, and the corresponding syscalls may be related to the target. According to the information of files `pkg/sentry/syscalls/linux/linux64.go` and `pkg/sentry/syscalls/linux/vfs2/vfs2.go` in gVisor code, we construct the mapping table of the functions and the corresponding syscalls. Second, we extract the related syscalls from the *reachable functions*. Note that not all these syscalls have close relevance to the target. To reduce false positives, in this rule, we only select the syscalls of the function that has the smallest distance to the target as the inference results.

Specialized Syscall Inference. The syscalls may have multiple parameters, and some of them may have complex structures and various values. To provide precise and semantically correct mutation, Syzkaller provides the syscall variants [27] by instantiating partial arguments of the original syscalls. For instance, `ioctl$FIONREAD` is a variant of syscall `ioctl`, which denotes the value of its parameter `request` is set as `FIONREAD` (the concrete value is 1074030207). When Syzkaller mutates this variant, the `request` parameter will not be modified. For original syscall `ioctl`, Syzkaller implements 411 syscall variants for

---

[①]For bug reproduction, if a user has the crash report, the eighth inference rule *stack trace based syscall inference* will be used.
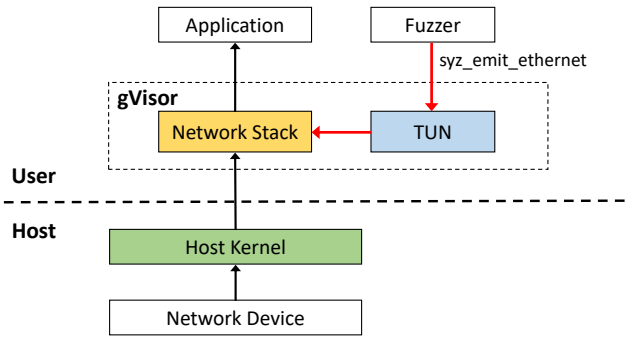
Fig. 4: The workflow of gVisor in processing network packets.

```
syz_emit_ethernet(                                      1
  len      len[packet],                                 2
  packet ptr[in, eth_packet],                           3
  frags    ptr[in, vnet_fragmentation, opt])            4
                                                        5
eth_packet {                                            6
  dst_mac   mac_addr                                    7
  src_mac   mac_addr                                    8
  vtag      optional[vlan_tag]                          9
  payload   eth_payload                                 10
} [packed]                                              11
                                                        12
eth_payload {                                           13
  eth2      eth2_packet                                 14
} [packed]                                              15
                                                        16
eth2_packet [                                           17
  generic   eth2_packet_generic                         18
  arp       eth2_packet_t[ETH_P_ARP, arp_packet]        19
  ipv4      eth2_packet_t[ETH_P_IP, ipv4_packet]        20
  ipv6      eth2_packet_t[ETH_P_IPV6, ipv6_packet]      21
  llc       eth2_packet_t[ETH_P_802_2, llc_packet]      22
  ... // 7 more union selections omitted                23
] [varlen]                                              24
                                                        25
ipv4_packet [                                           26
  generic ipv4_packet_t[flags[ipv4_types,int8],         27
                        array[int8]]
  tcp       ipv4_packet_t[const[IPPROTO_TCP,int8],       28
                        tcp_packet]
  udp       ipv4_packet_t[const[IPPROTO_UDP,int8],       29
                        udp_packet]
  icmp      ipv4_packet_t[const[IPPROTO_ICMP,int8],      30
                        icmp_packet]
  ... // 4 more union selections omitted                31
] [varlen]                                              32
```

Fig. 5: The parameters of syz_emit_ethernet.

it. If we successfully infer that the ioctl is related to a target but not be able to infer the concrete parameters, it is still challenging to reach this target.

In total, Syzkaller implements 948 syscall variants for 47 original syscalls. Thus, to reduce the input space, we have to infer the target related syscall variants. By conducting an in-depth analysis of the gVisor code, we find that the variant information can be extracted from the related code paths. Fig. 3 illustrates an example code that is related with ioctl. From the constants of the case statements, we can extract the concrete variants of ioctl. For instance, the code on line 5 is related with syscall variant ioctl$FIONREAD, and the code on line 7 is related with syscall variant ioctl$TCGETS. Thus, we infer the syscall variants by extracting the special constants from the code path towards the target. Notably, in the inference process, if an inferred syscall has variants, we then perform further inference about its variants, and add the syscall variants into the inference results. Based on this method, we can infer 127 variants for nine syscalls accurately. Table 1 presents the information about the nine syscalls.

Network Packet Processing Related Syscalls. gVisor implements a user-space network stack (i.e., netstack) to process the network packets between the host kernel and the container applications. Fig. 4 illustrates the how gVisor processes the network packets. In the practical scenario, the host kernel receives the network packets from the network device, and then transfers the packets to the netstack of gVisor. In essence, the inputs to trigger of netstack come from the network devices, not the upper syscalls. Nevertheless, deploying the real network devices to generate inputs to trigger the targets of netstack is not practical. To solve this problem, Syzkaller designs an extra syscall named syz_emit_ethernet. As red arrows in Fig. 4 shows, syz_emit_ethernet leverages the TUN, a virtual network device to directly inject network packets to netstack. Thus, we can leverage the syz_emit_ethernet syscall to test the targets of gVisor network stack code.

Nevertheless, the parameters of syz_emit_ethernet syscall are complex, which are related with many nested unions and structs. As Fig. 5 shows, there are three parameters of syz_emit_ethernet. Taking the second parameter packet ptr as example, packet ptr points to a struct type data structure, which consists of four members. The fourth member variable eth_payload is

a struct, and its member eth2_packet is an union. eth2_packet includes 12 members, and its third memeber ipv4 is also an union. ipv4_packet, a member of ipv4 is also an union, which consists of eight members. As a consequence, the parameters of syz_emit_ethernet are nested and complex. Only successfully inferring the target related syscall syz_emit_ethernet is not enough nor efficient. The whole input space of syz_emit_ethernet is vast due to the complex parameters.

To address this problem, we conduct in-depth analysis on the syz_emit_ethernet related targets. We find that some parameters of syz_emit_ethernet can be inferred based on the information of the targets. For instance, for the target function handleICMP that locates in the directory network/ipv4/icmp.go of gVisor, intuitively, it is related with ipv4 and icmp. Moreover, some parameters are mutually exclusive from each other. For instance, if the target is related with ipv4, it may have less relevance to ipv6. For the target function handleICMP, we can set the parameter eth2_packet as ipv4, and set the parameter ipv4_packet as icmp_packet, with high probability. In this way, we can reduce the input space efficiently. According to the different parameters, we implement ten syscall variants for syz_emit_ethernet. The detailed information of the ten syscall variants is shown in Table 2.

Virtual File System Related Syscalls. gVisor and other Unix-like kernels implement virtual file systems such as proc, devpts, etc. Different from the file systems such as ext4, FAT32, virtual file systems are virtual and do not locate in the disk space. The content of the virtual file systems resides in memory. The container applications use

TABLE 2: The variants of syscall `syz_emit_ethernet` in G-Fuzz.

Variants	Union Selections
syz_emit_ethernet$ipv4	eth2_packet=ipv4
syz_emit_ethernet$ipv4_tcp	eth2_packet=ipv4 && ipv4_packet=tcp
syz_emit_ethernet$ipv4_udp	eth2_packet=ipv4 && ipv4_packet=udp
syz_emit_ethernet$ipv4_icmp	eth2_packet=ipv4 && ipv4_packet=icmp
syz_emit_ethernet$ipv4_igmp	eth2_packet=ipv4 && ipv4_packet=igmp
syz_emit_ethernet$ipv6	eth2_packet=ipv6
syz_emit_ethernet$ipv6_tcp	eth2_packet=ipv6 && ipv6_packet=tcp
syz_emit_ethernet$ipv6_udp	eth2_packet=ipv6 && ipv6_packet=udp
syz_emit_ethernet$ipv6_icmp	eth2_packet=ipv6 && ipv6_packet=icmp
syz_emit_ethernet$arp	eth2_packet=arp

TABLE 3: Different Virtual File System Related syscalls.

Virtual Filesystem	Inferred Syscalls
devpts	openat$ptmx, syz_open_pts
eventfd	eventfd2, eventfd
kernfs	syz_open_procfs
pipefs	pipe, pipe2
signalfd	signalfd4, signalfd
timerfd	timerfd_create, timerfd_settime, timerfd_gettime

```
func pipe2(t *kernel.Task, addr usermem.Addr, flags int32)      1
    error {
    ...                                                          2
    if _, err := primitive.CopyInt32SliceOut(t, addr, fds);      3
        err != nil {
        for _, fd := range fds {                                 4
            if _, file := t.FDTable().Remove(t, fd); file        5
                != nil {
                file.DecRef(t)                                   6
            }                                                    7
        }                                                        8
        return err                                               9
    }                                                            10
    return nil                                                   11
}                                                                12
```

Fig. 6: An example of memory copy error handling code in gVisor.

the interfaces such as `open`, `read`, `write` to communicate with the kernel. Specifically, by setting different parameters, applications can request different file operations. If the targets reside in the virtual file system implementation code of gVisor, the corresponding syscalls to trigger them are the common interfaces.

Nevertheless, only knowing the interfaces is not enough to trigger these targets. These common interfaces can be invoked by much code. For instance, taking the function `masterFileDescription.Read` of `devpts` virtual file system as the target, the syscall to trigger this target is `read` and the its first parameter should be a `devpts` type file descriptor. We can use syscall `openat$ptmx` to generate a `devpts` related file descriptor. Thus, in addition, to infer the `read` syscall, we have to infer the corresponding parameter type. Based on the expert knowledge, we provide the inference rules for six types of the most used virtual file systems, which are shown in Table 3.

Error Handling Code Related Syscalls. There is much error handling code in OS kernels including gVisor, which is critical and may contain serious security problems [28], [29]. It is vital to test error handling code. However, the current mechanism of Syzkaller cannot effectively test the error handling code. For instance, Fig. 6 presents an example of memory copy error handling code in gVisor. The function `primitive.CopyInt32SliceOut` at line 3 is to copy the third parameter `fds` to the second parameter `addr`. colorred To trigger the target that resides in the code of line 4 to 9, the input sequence must make function `primitive.CopyInt32SliceOut` eturn an error. Syzkaller generates the concrete value of the syscall parameter based on its type. For this instance, the parameter `addr` is a pointer type, and Syzkaller tends to assign a valid memory address for `addr`. However, this target branch can not be triggered with a valid memory address.

To trigger the error handling code, intuitively, it should increase the probability of "causing an error". Here, we mainly focus on two common types of error handling code of gVisor: 1) memory related and 2) permission check related error handling code. For memory related error

handling code, we can add three syscalls: `mmap`, `munmap` and `mprotect` into the inferred results to make some memory invalid. Specifically, setting the parameter `prot` of syscalls `mmap` and `mprotect` as `PROT_NONE`, or using syscall `munmap` to release memory, can likely make the pointer type parameter points to the invalid memory.

For permission check related error handling code, we can add two syscalls: `setuid` and `setresuid` to change the current user of the running process. Thus, the syscalls after `setuid` or `setresuid` are more likely to have no access to the previously created resources, which increases the probability of permission check errors.

Seccomp Related Syscalls. *Seccomp* (Secure computing mode) is a security mechanism in Linux kernel, which is also adopted in gVisor. By setting the filter rules, *seccomp* restricts the syscalls that are allowed to be accessed by the current process. In particular, programmers can specify which system calls are permitted by writing *Berkeley Packet Filter (BPF)* programs. The *BPF* related code in gVisor resides in directory *pkg/bpf*, which is to parse and execute the *BPF* rules. However, these code are not directly executed when registering *BPF* rules, but are triggered by a hook mechanism afterward. Before subsequent syscalls of the current process can be executed, the hook will first execute all registered *BPF* rules to determine whether the syscall is allowed, which will trigger the code under the directory *pkg/bpf*. Thus, the *Function Call Chain based* inference rule cannot effectively find these registering syscalls. To tackle this problem, when testing this code as the target, we add the four *BPF* registering syscalls as the inference results, which are `prctl$PR_SET_SECCOMP`, `seccomp$SECCOMP_SET_MODE_STRICT`, `seccomp$SECCOMP_SET_MODE_FILTER` and `seccomp$SECCOMP_SET_MODE_FILTER_LISTENER`. These syscalls can register the *BPF* hook for subsequent syscalls or change the operating mode of seccomp of the current process. It is worth noting that there are more syscalls that are related to *BPF* mechanism, like `bpf`. Nevertheless, gVisor does not support these syscalls by directly returning capability errors. Therefore, we do not include them in the inference results.

Readiness Mechanism Code Related Syscalls. In order to achieve high-performance I/O operations, the Linux kernel proposes *epoll* mechanism to provide scalable I/O event notification. The applications do not

```
panic: Unknown syscall 165 error: strconv.ParseInt: parsing "": invalid syntax
goroutine 610 [running]:
panic(0xfc5280, 0xc0002162e0)
    ...
    gvisor.dev/gvisor/pkg/sentry/kernel.(*Task).doSyscallInvoke(0xc0005ff500, 0xa5,
0x20001d40, 0x20001d80, 0x20001dc0, 0x2000000, 0x20001e00, 0x0, 0x20001e00, 0x0)
    pkg/sentry/kernel/task_syscall.go:249 +0x165 fp=0xc00025fce0 sp=0xc00025fc58
pc=0x9cefa5
gvisor.dev/gvisor/pkg/sentry/kernel.(*Task).doSyscallEnter(0xc0005ff500, 0xa5, 0x20001d40,
0x20001d80, 0x20001dc0, 0x2000000, 0x20001e00, 0x0, 0xde0665, 0xc0004ae2e8)
    ...
```

Fig. 7: The stack trace information of a gVisor bug.

need to repeatedly query the status of the file descriptor to see whether I/O operation is possible, but instead, wait for the kernel to notify the state changes of the file descriptors. gVisor also implements this *epoll* mechanism, and adds a virtual file system level interface named `vfs.FileDescriptionImpl.Readiness`. All virtual file systems implement this interface to return whether the current file is readable or writable. With the understanding of this epoll mechanism, for Readiness function-related targets, that is, the target itself is implementing `vfs.FileDescriptionImpl.Readiness` interface or called by a Readiness function, we should add epoll-related syscalls into our inference results, including `pselect6`, `epoll_ctl$EPOLL_CTL_ADD`, `ppoll` and `poll`.

Stack Trace Based Syscall Inference. Bug reproduction is a typical application scenario of directed fuzzing. Compared to the general directed fuzzing scenarios that are only provided with the target locations, bug reproduction scenarios usually have more information, such as the stack trace information when the bug is triggered. Thus, we should make full use of this information when reproducing the bugs. Fig. 7 illustrates the stack trace and a PoC information of a gVisor bug. We can extract the bug related syscalls from its stack trace. For instance, the second parameter (i.e., 0xa5) of function `doSyscallInvoke` or `doSyscallEnter` represents the ID of syscall `mount`, and the PoC of this bug contains the syscall `mount`. Intuitively, for this bug, we can reproduce it effectively if we can infer its related syscalls.

Motivated by the above instance, for the target of bug reproduction, we infer its related syscalls from the stack trace (if given). First, similar to the function call chain based inference method, we extract the related syscalls from the functions of the stack trace. Second, we extract the related syscalls from the arguments of special functions such as `doSyscallInvoke` to find the corresponding syscalls. By incorporating the extracted syscalls into the inference results, G-Fuzz leverages the information from the stack trace to facilitate bug reproduction.

3.4 The Utilization of the Inferred Syscalls

In directed fuzzing process, the inferred syscalls are utilized in the mutation process to generate new inputs. We define the probability that whether we choose to use the inferred syscalls in the mutation as p. The value of p decreases linearly during the fuzzing process. Assume the maximum value of p is p_{max}, and the minimum value of p is p_{min}. The fuzzing timeout is T_{fuzz}. The value of p at the current fuzzing time t is:

$$p = p_{max} - \frac{p_{max} - p_{min}}{T_{fuzz}} * t \qquad (1)$$

In our experiment, we set $p_{max} = 0.9$ and $p_{min} = 0.1$. At the beginning of directed fuzzing, the probability of utilizing the inferred syscalls is 90%. With the fuzzing time increases, the probability decreases linearly until 10%. The rationale behind this design choice is that if we cannot trigger the target in a relatively long time with the inferred syscalls, the inference results may have false positives and further mislead the fuzzing process. Thus, we should decrease the probability of using the inferred syscalls.

The inferred syscalls are used to insert into a selected seed to generate the new inputs. To make efficient use of the inferred syscalls and generate semantically correct inputs, we propose two methods in the mutation stage: *Selection Probability Scheduling Strategy* and *Insert Order Scheduling Strategy*.

Selection Probability Scheduling Strategy. There may exist false positives in the inference results. In other words, some inferred syscalls may not be related to the target in fact, which may reduce the efficiency of directed fuzzing. To alleviate the impacts of the false positives, we propose a strategy to dynamically adjust the selection probability of each inferred syscall in the mutation process.

Assume the set of the inferred syscalls for the given target is $S = \{s_1, s_2, ..., s_n\}$, where n is the number of all inferred syscalls. The initial weight W of each inferred syscall is one, which is equal. That is, $W_{S_1} = W_{S_2} = ... = W_{S_n} = 1$.

During the fuzzing process, the inputs which can trigger new paths will be saved in a *global seed queue*. For the new saved input, we will determine whether its distance is shorter than its parent seed. If so, we will save it in another seed queue named *shorter distance queue*. Intuitively, the seed in *shorter distance queue* may contain the syscalls that can be helpful to reach the target. For each inferred syscall, we count its frequency in *shorter distance queue*. An incorrect inferred syscall is highly unlikely to appear in this queue. Assume at fuzzing time t, the frequency of an inferred syscall s_i is $F_{(s_i,t)}$. At time t, the inferred syscall s_i's weight $W_{s_i}(t)$ is $F_{(s_i,t)} + 1$. The selection probability of the syscall s_i is calculated by formula 2.

$$P_{s_i}(t) = \frac{W_{s_i}(t)}{\sum_{j=1}^{n} W_{s_j}(t)} \qquad (2)$$

As the fuzzing goes on, the probability of selecting the correct inferred syscalls increases, and the probability of selecting the incorrect inferred syscalls decreases. In this way, G-Fuzz can effectively mitigate the negative impacts of incorrect inference results.

Insert Order Scheduling Strategy. The order of the syscalls has a significant impact on the semantic of the whole input. There may exist dependencies between different syscalls. MoonShine [30] classifies the dependency relationships between different syscalls into two types: *explicit* and *implicit* dependencies. The explicit dependencies represent that the return value is used as an argument of another syscall. The implicit dependencies represent that there are shared data structures among syscalls. Thus, to generate the semantically correct inputs as possible, the syscalls dependent on others should be placed at the front position of the entire sequence.

MoonShine focuses on generating the initial seeds with offline static analysis. In contrast, we aim to generate inputs

by mutating seeds during the fuzzing process, which is an online and time-sensitive task. To provide fast mutation, we mainly consider the *explicit* dependencies. We regard the syscalls that generate the dependent return values as the "producers" and regard the syscalls that receive the dependent values as the "consumers". Intuitively, the "producer" syscalls should be inserted in front of the "consumer" syscalls. To this end, we use the function `biasedRand` provided by Syzkaller to generate the insert index for "consumer" syscalls. In specific, `biasedRand(n, k)` is to randomly generate a number, which conforms that the probability of outputting n-1 is k times higher than the probability of outputting 0. We set k as 5 in the experiments. For the "producer" syscall, we use `n - biasedRand(n, k)` to generate its insert index. In this way, the "producer" syscalls tend to be inserted in front of the "consumer" syscalls.

3.5 Exploration and Exploitation Dynamic Switch

The core idea of directed greybox fuzzing is using distance information to guide the fuzzer to constantly generate test cases closer to the target until triggering it. However, distance information has limitations in guiding directed fuzzing. As we discussed in §3.2, distance information has low precision. Despite the issues caused by distance calculation methods, existing static analysis technique cannot identify the indirect calls with no error [22]. In addition, distance cannot be able to measure the difficulty to reach the target. Thus, it is necessary to make reasonable use of the distance information.

One practical solution is to incorporate the *exploitation* with *exploration*. Here the *exploitation* represents the typical "directed fuzzing", which focuses on testing the shortest or closer paths to the target. The *exploration* represents the typical "coverage-based fuzzing", which focuses on exploring the new paths. The *exploration* can alleviate the errors of the distance and can avoid directed fuzzing trapping into the local optimal. Nevertheless, overusing *exploration* makes directed fuzzing degenerate into the coverage-based fuzzing, which is not efficient.

Thus, how to coordinate the *exploitation* and *exploration* to amplify their advantages in directed fuzzing is critical. AFLGo [7] adopts a fixed time-wise splitting method to balance the exploration and exploitation phases. Specifically, for a 24-hour fuzzing experiment, AFLGo sets the exploration time as 20 hours and the exploitation time as 4 hours. However, this fixed setting may not suit all the conditions in directed fuzzing, such as testing different programs or targets. Paper [15] shows that the performance of AFLGo varied much with different settings in exploration time. Therefore, the solution of AFLGo is not satisfying. Other fuzzing work such as EcoFuzz [31] also studies the *exploitation* and *exploration* trade-off problem, whereas it focuses on solving the seed power scheduling problem, not the directed fuzzing.

To tackle this problem, we propose a *exploration and exploitation dynamic switch* method, which coordinates the two phases adaptively according to the current state of fuzzing. When one phase does not make any contribution to fuzzing for a time threshold, gVisor will switch to the

other phase, and vice versa. The detailed workflows of this method are the following.

At the start of directed fuzzing, G-Fuzz first enters the initial phase, generating and executing the initial inputs. The inputs that can discover new paths will be saved in the seed pool. After this phase, G-Fuzz first enters the closer seed exploitation phase. In this phase, G-Fuzz randomly selects m seeds from the seed pool and selects the top k seeds with the shortest distance. Note that the distance of a seed is the shortest distance of its executed basic blocks. The shortest k seeds are mutated to generate new inputs. Then, G-Fuzz executes the new inputs. For the inputs that trigger the new paths, G-Fuzz saves them in the seed pool. During the fuzzing process, G-Fuzz monitors whether the current phase gets into a "stuck" state. If so, G-Fuzz will switch into the other phase. For the closer seed exploitation phase, the "stuck" state is not finding any new paths of the reachable set over the time threshold T_a. When G-Fuzz switches into the path exploration phase, it does not consider the distance of each seed. In specific, it selects seeds from the seed pool according to the coverage of each seed. The seed with more coverage has a higher probability of being selected. During this phase, if a seed finds new paths, before saving it in the seed pool, it will be directly mutated to generate the new inputs. The "stuck" state of this phase is not finding any new paths in time threshold T_b. Where T_a and T_b are user-defined. In our experiments, we set T_a to five minutes, and T_b to ten minutes.

4 IMPLEMENTATION OF G-FUZZ

G-Fuzz contains two components: static analysis and directed fuzzing. The functionalities of the static analysis include CG/CFG extraction, reachability analysis, distance calculation and target related syscalls inference. We implement the CG extraction based on `go-callvis` [26]. For the CFG of gVisor, we first use `GoLLVM` [32] to compile gVisor into LLVM IR, an intermediate representation. Then, we implement an LLVM Pass to construct the target related inter-procedural CFG. The Pass has 1,238 lines of C++ code. The distance calculation of G-Fuzz is implemented with 313 lines of Python code. The target related syscalls inference is written with 212 lines of Python code. The directed fuzzing component of G-Fuzz is implemented based on the state-of-the-art OS kernel fuzzer Syzkaller (commit ID=9d751681c). We add more than 1,500 lines of Go language code, comparing with the origin Syzkaller.

5 EVALUATION

In this section, we mainly evaluate the performance of G-Fuzz in directed fuzzing for gVisor.

Basic Settings. As there exists randomness in fuzzing experiments, following the guidance in paper [33], we conduct each experiment with 20 repetitions. For general target testing and patch testing, the fuzzing time is 24 hours. For bug reproduction, as it is more difficult, the fuzzing time is 72 hours.

TABLE 4: The performance of G-Fuzz, Syz-Go and Syzkaller in general target testing.

Target ID	Fuzzer	runs	μTTE (h)	Speedup	$\hat{A}_{12}$	p-value
#1	G-Fuzz	**18**	**6.79**	-	-	-
	Syz-Go	**18**	12.43	1.83	0.79	<0.01
	Syzkaller	12	17.26	2.54	0.87	<0.01
#2	G-Fuzz	**20**	**0.92**	-	-	-
	Syz-Go	16	15.06	16.39	0.99	<0.01
	Syzkaller	7	20.57	22.38	1.00	<0.01
#3	G-Fuzz	**20**	**0.15**	-	-	-
	Syz-Go	18	14.23	97.46	1.00	<0.01
	Syzkaller	8	19.27	131.96	1.00	<0.01
#4	G-Fuzz	19	**1.36**	-	-	-
	Syz-Go	**20**	5.40	3.96	0.95	<0.01
	Syzkaller	14	15.42	11.31	0.96	<0.01
#5	G-Fuzz	**20**	**0.20**	-	-	-
	Syz-Go	**20**	5.40	26.99	0.99	<0.01
	Syzkaller	17	11.55	57.80	1.00	<0.01
#6	G-Fuzz	**20**	**0.18**	-	-	-
	Syz-Go	19	6.05	33.84	1.00	<0.01
	Syzkaller	**20**	8.23	46.09	1.00	<0.01
#7	G-Fuzz	**9**	**18.02**	-	-	-
	Syz-Go	3	21.21	1.18	0.63	0.044
	Syzkaller	5	21.28	1.18	0.61	0.077
#8	G-Fuzz	**19**	**4.69**	-	-	-
	Syz-Go	4	20.28	4.33	0.91	<0.01
	Syzkaller	2	22.91	4.89	0.95	<0.01
#9	G-Fuzz	**10**	**14.49**	-	-	-
	Syz-Go	2	23.28	1.61	0.72	<0.01
	Syzkaller	0	24.00	1.66	0.75	<0.01
#10	G-Fuzz	**19**	**4.28**	-	-	-
	Syz-Go	10	17.85	4.17	0.91	<0.01
	Syzkaller	6	20.30	4.74	0.93	<0.01
#11	G-Fuzz	**12**	**15.20**	-	-	-
	Syz-Go	8	20.78	1.37	0.67	0.027
	Syzkaller	5	19.92	1.31	0.66	0.032
#12	G-Fuzz	**13**	**11.20**	-	-	-
	Syz-Go	7	19.85	1.77	0.72	<0.01
	Syzkaller	5	21.98	1.96	0.78	<0.01
#13	G-Fuzz	**10**	**13.78**	-	-	-
	Syz-Go	1	23.01	1.67	0.73	<0.01
	Syzkaller	0	24.00	1.74	0.75	<0.01
#14	G-Fuzz	**16**	**9.36**	-	-	-
	Syz-Go	12	15.64	1.67	0.71	0.012
	Syzkaller	12	14.43	1.54	0.61	0.111
#15	G-Fuzz	**20**	**2.26**	-	-	-
	Syz-Go	15	11.68	5.16	0.89	<0.01
	Syzkaller	15	11.61	5.13	0.91	<0.01
#16	G-Fuzz	**19**	**1.57**	-	-	-
	Syz-Go	6	19.63	12.53	0.97	<0.01
	Syzkaller	9	18.21	11.63	0.96	<0.01
#17	G-Fuzz	**20**	**1.46**	-	-	-
	Syz-Go	16	14.86	10.15	0.99	<0.01
	Syzkaller	16	13.98	9.55	0.95	<0.01
#18	G-Fuzz	**8**	**17.45**	-	-	-
	Syz-Go	7	20.42	1.17	0.56	0.252
	Syzkaller	6	19.32	1.11	0.56	0.247
#19	G-Fuzz	**16**	**8.35**	-	-	-
	Syz-Go	13	14.23	1.70	0.74	<0.01
	Syzkaller	12	16.29	1.95	0.73	<0.01
#20	G-Fuzz	**19**	**2.89**	-	-	-
	Syz-Go	8	18.78	6.51	0.93	<0.01
	Syzkaller	16	14.39	4.99	0.92	<0.01
#21	G-Fuzz	**14**	**10.20**	-	-	-
	Syz-Go	4	20.68	2.03	0.78	<0.01
	Syzkaller	12	17.07	1.67	0.73	<0.01
#22	G-Fuzz	**15**	**12.16**	-	-	-
	Syz-Go	7	20.39	1.68	0.76	<0.01
	Syzkaller	9	18.16	1.49	0.73	<0.01
#23	G-Fuzz	**17**	**14.92**	-	-	-
	Syz-Go	8	19.64	1.32	0.72	<0.01
	Syzkaller	11	17.94	1.20	0.64	0.069
#24	G-Fuzz	**20**	**0.31**	-	-	-
	Syz-Go	10	17.68	57.17	1.00	<0.01
	Syzkaller	15	13.54	43.81	1.00	<0.01
#25	G-Fuzz	**3**	**21.46**	-	-	-
	Syz-Go	1	23.90	1.11	0.55	0.138
	Syzkaller	0	24.00	1.12	0.57	0.040
#26	G-Fuzz	**2**	**23.52**	-	-	-
	Syz-Go	1	23.94	1.02	0.53	0.267
	Syzkaller	0	24.00	1.02	0.55	0.081

TABLE 5: The performance of G-Fuzz, Syz-Go and Syzkaller in patch testing.

Patch ID	Fuzzer	runs	μTTE (h)	Speedup	$\hat{A}_{12}$	p-value
#1	G-Fuzz	20	**1.16**	-	-	-
	Syz-Go	20	3.41	2.95	0.83	<0.01
	Syzkaller	20	2.11	1.82	0.74	<0.01
#2	G-Fuzz	20	**1.01**	-	-	-
	Syz-Go	20	2.40	2.38	0.77	<0.01
	Syzkaller	20	2.20	2.18	0.78	<0.01
#3	G-Fuzz	**20**	**0.07**	-	-	-
	Syz-Go	**20**	4.42	61.94	1.00	<0.01
	Syzkaller	18	6.90	96.73	1.00	<0.01
#4	G-Fuzz	**18**	**3.75**	-	-	-
	Syz-Go	14	17.11	4.56	0.89	<0.01
	Syzkaller	3	21.99	5.86	0.93	<0.01
#5	G-Fuzz	**18**	**6.08**	-	-	-
	Syz-Go	15	14.50	2.38	0.85	<0.01
	Syzkaller	12	14.98	2.46	0.77	<0.01
#6	G-Fuzz	**18**	**12.06**	-	-	-
	Syz-Go	7	19.98	1.66	0.81	<0.01
	Syzkaller	2	22.98	1.91	0.91	<0.01
#7	G-Fuzz	**20**	**1.24**	-	-	-
	Syz-Go	**20**	5.75	4.64	0.91	<0.01
	Syzkaller	17	12.03	9.72	0.98	<0.01
#8	G-Fuzz	17	**5.24**	-	-	-
	Syz-Go	**19**	6.20	1.18	0.73	<0.01
	Syzkaller	15	14.05	2.68	0.85	<0.01
#9	G-Fuzz	20	**0.33**	-	-	-
	Syz-Go	20	2.75	8.42	1.00	<0.01
	Syzkaller	20	4.28	13.09	1.00	<0.01
#10	G-Fuzz	20	**0.35**	-	-	-
	Syz-Go	20	1.54	4.45	0.95	<0.01
	Syzkaller	20	1.80	5.19	0.94	<0.01
#11	G-Fuzz	19	2.81	-	-	-
	Syz-Go	**20**	1.52	0.54	0.49	0.473
	Syzkaller	**20**	**1.50**	0.53	0.49	0.473
#12	G-Fuzz	18	4.53	-	-	-
	Syz-Go	**20**	**2.60**	0.57	0.62	0.104
	Syzkaller	**20**	4.67	1.03	0.69	0.023
#13	G-Fuzz	20	**0.19**	-	-	-
	Syz-Go	20	2.25	11.86	1.00	<0.01
	Syzkaller	20	3.68	19.40	1.00	<0.01
#14	G-Fuzz	20	**0.42**	-	-	-
	Syz-Go	20	1.23	2.92	0.83	<0.01
	Syzkaller	20	1.52	3.60	0.92	<0.01
#15	G-Fuzz	20	**0.13**	-	-	-
	Syz-Go	20	1.98	15.39	1.00	<0.01
	Syzkaller	20	5.25	40.73	1.00	<0.01
#16	G-Fuzz	**20**	**0.26**	-	-	-
	Syz-Go	**20**	2.91	11.08	0.99	<0.01
	Syzkaller	19	6.32	24.05	1.00	<0.01
#17	G-Fuzz	**20**	**4.60**	-	-	-
	Syz-Go	13	13.73	2.98	0.81	<0.01
	Syzkaller	11	17.33	3.76	0.89	<0.01
#18	G-Fuzz	20	**0.21**	-	-	-
	Syz-Go	20	4.62	21.85	1.00	<0.01
	Syzkaller	20	3.53	16.70	1.00	<0.01
#19	G-Fuzz	20	**1.47**	-	-	-
	Syz-Go	20	3.98	2.70	0.76	<0.01
	Syzkaller	20	3.90	2.65	0.82	<0.01
#20	G-Fuzz	20	**1.26**	-	-	-
	Syz-Go	20	5.57	4.42	0.94	<0.01
	Syzkaller	20	5.92	4.70	0.94	<0.01
#21	G-Fuzz	5	21.07	-	-	-
	Syz-Go	**7**	**21.01**	1.00	0.46	0.297
	Syzkaller	0	24.00	1.14	0.62	<0.01
#22	G-Fuzz	**3**	**22.85**	-	-	-
	Syz-Go	0	24.00	1.05	0.57	0.040
	Syzkaller	0	24.00	1.05	0.57	0.040

5.1 Experimental Settings

Compared Fuzzers. We compare G-Fuzz with two fuzzers: Syzkaller and Syz-Go. Syzkaller is the state-of-the-art kernel fuzzer. We implement Syz-Go by adopting the method of AFLGo on Syzkaller. Specifically, as the time overhead of the original AFLGo implementation in distance calculation is high, we use the same distance calculation implementation in Syz-Go as G-Fuzz.

For the directed fuzzing part of Syz-Go, we implement the simulated annealing-based power scheduling algorithm of AFLGo in it. Following the original design of AFLGo [14], for the 24-hour experiment, the exploration time of Syz-Go is 20 hours, and the exploitation time is 4 hours. For the 72-hour experiment, the exploration time of Syz-Go is 60 hours, and the exploitation is 12 hours.

Evaluation Metrics. Following the methods of the state-of-the-art directed fuzzing papers [7], [8], we mainly use *runs* and *Time-to-Exposure (TTE)* as the evaluation metrics. Specifically, *runs* represents the times that a fuzzer triggers the target in multiple repeated experiments. *TTE* is the first time that a fuzzer triggers the target, and μTTE is the arithmetic average of *TTE* in multiple repeated experiments. In specific, if a fuzzer does not trigger the target over the fuzzing timeout (e.g., 24 hours), the *TTE* of this experiment is regarded as the timeout.

Moreover, we use statistical metrics such as $\hat{A}_{12}$ [34] and p-value to provide more comprehensive evaluations. $\hat{A}_{12}$ is the metric to measure the effect size, that is, the probability of one group of samples perform better than the other. The p-value is to measure whether the difference between the two groups is real or due to randomness. If the p-value is less than 0.05 (or 0.01), there is a significant difference between the two groups. We use the Mann-Whitney U test, a non-parametric statistical test, to calculate the p-value.

Initial Seeds and Environment. For Syzkaller and Syz-Go, the initial seeds are the same, generated by the original design of Syzkaller. For G-Fuzz, the initial seeds are generated according to the inferred target related syscalls. We conduct all the experiments on several servers with the same settings. Each server has 20 Intel Xeon E5-2650 V4 CPUs, and the OS is 64-bits Ubuntu 16.04 LTS.

5.2 General Target Testing

In this scenario, the targets are the arbitrary locations of gVisor. We randomly select 50 code sites as the targets. Among the 50 targets, there are 24 targets that no fuzzer can find with the 24-hour × 20 repetition evaluation. Table 4 shows the performance of gVisor, Syz-Go and Syzkaller in directed fuzzing for the 26 targets.

From Table 4, we can get the following observations. (1) It is evident that G-Fuzz outperform both Syzkaller and Syz-Go in discovering the targets. In terms of $\hat{A}_{12}$ metric, G-Fuzz achieves more than 0.5 $\hat{A}_{12}$ values on all targets comparing to Syz-Go and Syzkaller. Specifically, G-Fuzz achieves more than 0.71 $\hat{A}_{12}$ on 21 targets, except for target #7, #11, #18, #25 and #26. $\hat{A}_{12}$ value is more than 0.71 means that the probability of one group performs better than the other is large [34]. For 20 targets, the p-value of G-Fuzz between Syz-Go and Syzkaller is less than 0.01, demonstrating the significantly better performance of G-Fuzz. (2) G-Fuzz has more

stable performance than Syz-Go and Syzkaller. According to the *runs* column, G-Fuzz has the highest values in 25 targets, except for target #4. (3) G-Fuzz accelerates the performance of directed fuzzing by a large margin. For instance, G-Fuzz only takes 0.92 hours in triggering target #2, whereas Syz-Go and Syzkaller need to use more than 12 hours. More importantly, for target #25 and #26, G-Fuzz can trigger them while Syzkaller cannot. From the *speedup* column, we can see that the max *speedup* that G-Fuzz can achieve is more than 130 (target #3). For seven targets, the *speedup* value of G-Fuzz is more than 10. In summary, the performance of G-Fuzz in testing the general targets of gVisor is stable and efficient.

5.3 Patch Testing

The targets of patch testing are the code modified by the patch commits. In this scenario, we select 50 code locations modified by the patch commits as the targets. Among the 50 patch targets, 28 of them are not found by any fuzzer with the 24-hour × 20 repetition evaluation. Table 5 shows the performance of G-Fuzz, Syzkaller and Syz-Go in directed fuzzing for the rest of 22 patch targets.

From Table 5, we can get the following observations. (1) Under the *runs* metrics, G-Fuzz has the best performance among the compared fuzzers on 18 targets, which demonstrates the effectiveness of G-Fuzz. (2) In terms of μTTE and *speedup* metrics, G-Fuzz takes the least amount of time in triggering 20 targets, except for target #11 and #12. Moreover, G-Fuzz can boost the performance of patch testing up to 96.73 times. (3) As for the $\hat{A}_{12}$ metric, G-Fuzz has more than 0.5 $\hat{A}_{12}$ value on 20 targets and has more than 0.71 $\hat{A}_{12}$ value on 18 targets. (4) For over 90 percent of targets (18/20), the p-value of G-Fuzz is less than 0.01. In summary, G-Fuzz outperforms Syz-Go and Syzkaller in patch testing.

5.4 Bug Reproduction

Bug reproduction is a significant application scenario of directed fuzzing. Assuming that we have already known the information about the bug, such as the vulnerable locations or the stack trace, etc., but do not have the PoC to validate the bug. The goal of directed fuzzing is to generate the PoC that can trigger the bug. In this scenario, we select the public gVisor bugs as the targets from the official Syzbot website [35]. For the public bugs, we first use the provided PoCs to validate whether the bugs can be reproduced. Finally, we select 30 bugs as the targets. It needs to be emphasized that we only use the provided PoCs when selecting target bugs. In the experiment, we assume that we do not have the PoCs and the goal of directed fuzzing is to generate the PoCs. For the 30 bugs, only 11 of them can be successfully triggered by at least one fuzzer within 72 hours with 20 repetitions. Table 6 presents the performance of G-Fuzz, Syz-Go and Syzkaller in bug reproduction.

From Table 6, we can observe the following facts. (1) For 11 bugs, G-Fuzz successfully trigger all of them, whereas Syz-Go fails to trigger two of them, and Syzkaller fails to trigger two of them. (2) In terms of the *runs* metric, G-Fuzz achieves the highest number for all bugs, which is significantly better than Syz-Go and Syzkaller. (3) The speed

TABLE 6: The performance of G-Fuzz, Syz-Go and Syzkaller in bug reproduction.

Bug ID	Fuzzer	runs	μTTE (h)	Speedup	$\hat{A}_{12}$	p-value
#1	G-Fuzz	20	3.91	-	-	-
	Syz-Go	20	13.53	3.46	0.88	<0.01
	Syzkaller	19	15.49	3.96	0.85	<0.01
#2	G-Fuzz	20	0.08	-	-	-
	Syz-Go	20	0.84	10.74	0.99	<0.01
	Syzkaller	20	2.09	26.82	1.00	<0.01
#3	G-Fuzz	3	65.37	-	-	-
	Syz-Go	2	66.68	1.02	0.52	0.345
	Syzkaller	3	69.71	1.07	0.51	0.431
#4	G-Fuzz	8	61.87	-	-	-
	Syz-Go	1	70.32	1.14	0.67	<0.01
	Syzkaller	1	71.22	1.15	0.68	<0.01
#5	G-Fuzz	4	62.16	-	-	-
	Syz-Go	2	69.61	1.12	0.56	0.175
	Syzkaller	1	71.70	1.15	0.58	0.069
#6	G-Fuzz	3	67.50	-	-	-
	Syz-Go	1	69.72	1.03	0.55	0.162
	Syzkaller	0	72.00	1.07	0.57	0.040
#7	G-Fuzz	6	63.30	-	-	-
	Syz-Go	5	60.60	0.96	0.51	0.466
	Syzkaller	3	65.75	1.04	0.57	0.163
#8	G-Fuzz	17	15.42	-	-	-
	Syz-Go	10	48.87	3.17	0.84	<0.01
	Syzkaller	5	62.27	4.04	0.89	<0.01
#9	G-Fuzz	7	52.44	-	-	-
	Syz-Go	0	72.00	1.37	0.68	<0.01
	Syzkaller	1	70.48	1.34	0.66	<0.01
#10	G-Fuzz	6	57.73	-	-	-
	Syz-Go	0	72.00	1.25	0.65	<0.01
	Syzkaller	2	68.24	1.18	0.60	0.054
#11	G-Fuzz	8	56.93	-	-	-
	Syz-Go	5	63.46	1.11	0.57	0.186
	Syzkaller	0	72.00	1.26	0.70	<0.01

of G-Fuzz in discovering the bugs is faster than both Syz-Go and Syzkaller. According to the *speedup* metric, G-Fuzz can boost the performance of bug reproduction up to 26.82 (target #2). (4) Compared to Syz-Go and Syzkaller, G-Fuzz has over 0.5 $\hat{A}_{12}$ value in reproducing all 11 bugs. Besides, for seven bugs, the p-value of G-Fuzz is less than 0.01. In summary, the above observations prove that G-Fuzz has outstanding performance in reproducing bugs.

6 FURTHER ANALYSIS

To provide comprehensive evaluations, in this section, we conduct further experiments and analysis on the effectiveness of each core methods of G-Fuzz, as well as ablation studies. Moreover, we present the deployment and the application of G-Fuzz in real-world.

6.1 Static Distance Calculation Method

Below, we measure the performance of the static distance calculation method of G-Fuzz from two aspects: time overhead and precision.

Time Overhead. To measure the time overhead of G-Fuzz in distance calculation, we randomly select six versions of gVisor. We randomly select a code location as the target for each version to calculate the static distance for the rest code of gVisor. We compare the G-Fuzz with AFLGo. Table 7 presents the average time overhead of AFLGo and G-Fuzz in static distance calculation with three repetitions. For the six targets, on average, AFLGo spends 68,346 seconds (18.9 hours) in calculating the static distance, while G-Fuzz only

spends 97.1 seconds. Compared to AFLGo, G-Fuzz reduces the time overhead in distance calculation drastically. The third column of Table 7 shows the reachable function ratio of each target. Among the six targets, the highest ratio is 18.90%, and four of them only have less than 0.2% reachable ratios. Note that as we use *RTA* method to identify the indirect calls, there is almost no false negative in the result and the number of ground truth value of reachable function ratio may be even smaller/less. This observation shows that the target related code only accounts for a tiny fraction. The time of calculating static distance for the unrelated code is unnecessary and wasteful. Thus, G-Fuzz's reachability analysis based distance calculation can effectively reduce the time overhead.

In addition, we conduct a time profiling experiment on AFLGo. We find that the find_nodes function of AFLGo takes more than 50% time, and this function has been invoked almost two billion times on average. Given a string of a basic block (i.e., file name and the line number), the functionality of find_nodes function is to find its corresponding node object in the current CFG. Nevertheless, find_nodes function will be called for each basic block of all functions, whereas most of the traversed basic blocks do not belong to the current CFG. The complexity of this implementation is $O(N^2)$, where N is the number of basic blocks. The complexity can be reduced to $O(N)$ by only traversing basic blocks of the current CFG.

Precision of Distance. In addition to reducing the time overhead in distance calculation, G-Fuzz improves the distance's precision, compared with AFLGo. Based on the discussion in §3.2, there are two main issues of AFLGo in the precision of distance: (1) AFLGo does not consider the indirect calls. (2) AFLGo uses the coarse-grained function level distance to approximate the basic block level distance. Next, we discuss the impact of these two issues on directed fuzzing and whether G-Fuzz can solve them.

For issue (1), Table 8 shows the number of indirect calls of gVisor with six different versions. We can observe that the ratio of the indirect calls to the total calls ranges from 4.89% to 5.24%. Although the ratio of the indirect calls is not very large, it may cause missing edges. In particular, the absence of some critical edges will cause many errors in distance calculation. The sixth and seventh columns of Table 8 lists the number of target related functions (i.e., have the static distance) that can be found by AFLGo and G-Fuzz. We can observe that the number of target related functions decreases by a large margin if the indirect calls are not considered, which makes the fuzzer fail to find some important seeds. For issue (2), we present frequency distribution of the number of basic blocks within a gVisor function in Fig. 8. We can observe that the number of basic blocks has a wide range. Most of the gVisor functions have only one basic block. Some functions even have more than 20 basic blocks. The largest number of basic blocks that a gVisor function has is 162. Thus, instead of using a constant to approximate, we provide a fine-grained (i.e., basic block level) distance information in the implementation of G-Fuzz. Moreover, compared to AFLGo, G-Fuzz uses *RTA* to discover the indirect calls and calculates the basic block level distance with a low time overhead, effectively solves the two issues.

TABLE 7: The time overhead of AFLGo and G-Fuzz in distance calculation.

Commit ID	Target	Reachable Function Ratio	Time(s) of Static Distance Calculation	
			AFLGo	G-Fuzz
cdf49c44	pkg/tcpip/stack/linkaddrcache.go:189	18.88%	76,275	102.0
c564293b6	pkg/sentry/syscalls/linux/vfs2/splice.go:398	0.16%	60,507	92.8
76da673a	pkg/tcpip/network/ipv4/igmp.go:158	18.90%	76,233	102.6
8b9cb36d1	pkg/sentry/syscalls/linux/vfs2/splice.go:147	0.16%	60,339	91.6
55332aca9	pkg/sentry/vfs/file_description.go:828	0.17%	75,937	103.0
7f89a26e1	pkg/sentry/kernel/fd_table.go:369	0.19%	60,782	90.8
Average			68,346	97.1

TABLE 8: The impact of the indirect calls on the reachable sets of gVisor.

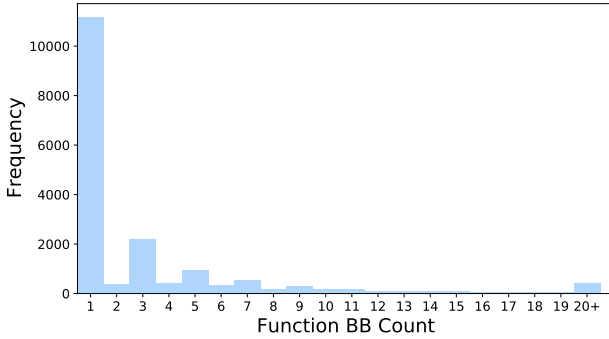| Commit ID	Target	#Call Sites	#Indirect Call	Indirect Call Ratio	# Target Related Functions	
					AFLGo	G-Fuzz
cdf49c44	pkg/tcpip/stack/linkaddrcache.go:189	79,048	4,142	5.24%	12	3,376
c564293b6	pkg/sentry/syscalls/linux/vfs2/splice.go:398	70,725	3,467	4.90%	1	24
76da673a	pkg/tcpip/network/ipv4/igmp.go:158	79,047	4,131	5.23%	1	3,376
8b9cb36d1	pkg/sentry/syscalls/linux/vfs2/splice.go:147	70,492	3,448	4.89%	1	24
55332aca9	pkg/sentry/vfs/file_description.go:828	78,922	4,120	5.22%	1	30
7f89a26e1	pkg/sentry/kernel/fd_table.go:369	70,872	3,476	4.90%	2	29

Fig. 8: The frequency distribution of the number of basic blocks within a function of gVisor.

Fig. 9: Ablation study 1: static distance calculation optimization.

Ablation Study. We conduct ablation study on the static distance calculation optimization. we compare G-Fuzz to G-Fuzz-func-dis (G-Fuzz that uses function level distance multiplied by a constant 10 to approximate basic block level distance) and Syzkaller in directed fuzzing 10 targets, with five repetitions. Fig. 9 shows the experimental results. We can obverse that, as a whole, G-Fuzz uses significantly less time in triggering the targets than G-Fuzz-func-dis and Syzkaller. G-Fuzz outperforms G-Fuzz-func-dis in 8 out of 10 targets (except for target 3 and target 10), and outperforms Syzkaller in 10 targets. Therefore, the experimental results demonstrate the effectiveness of the distance calculation optimization of G-Fuzz in directed fuzzing.

6.2 Target Related Syscall Inference and Utilization

We evaluate this method from two aspects: the precision of the inference and the effectiveness of the utilization.

Precision of Inference. To verify whether the inferred target related syscalls can help to trigger the targets, we compared the syscalls of the PoC[1] to the inference results. As there may exist syscalls in the PoC that have no contribution to triggering the target, we perform *minimization* operations on the PoC to remove redundant syscalls. In particular, we delete each syscall of a PoC in turn to obtain

a pruned PoC and execute it. If the pruned PoC can still trigger the target, then the deleted syscall will be considered redundant. We can obtain the *minimized PoC* by removing all redundant syscalls. We define the precision of the inference as the number of inferred syscalls that are contained in the *minimized PoC* divided by the number of all inferred syscalls.

Table 9 and Table 10 present the inferred syscalls and the precision for the general targets and the patch testing targets, respectively. For the general target #2, #18, and patch #11, #22, we cannot reproduce the PoC, and their *minimized PoC* is not presented in the table. The inferred syscalls that are contained in the *minimized PoC* is presented with bold.

We can observe that for most of the targets, the precision of inference is 100%, which means that all the inferred syscalls are necessary for triggering these targets. None of the target's inference precision is zero, which demonstrates that at least one inferred syscall could contribute to finding the target. It needs to emphasize that G-Fuzz provides solutions for mitigating the impacts caused by the false positives and negatives of the inference. For false positives, G-Fuzz uses a strategy (3.4) to reduce the selection probabilities of the false inferred syscalls. For false negatives, G-Fuzz reduces it by combining directed fuzzing with path exploration (3.5). Therefore, G-Fuzz can effectively improve the performance of directed fuzzing.

Effectiveness of Utilization. There may exist false pos-

[1]For the convenience of expression, here we use PoC to refer to the inputs that can trigger the target.

TABLE 9: The inferred syscalls for the general targets.

Target ID	Inferred Syscalls	Precision
#1	**syz_emit_ethernet**	100%
#2	syz_emit_ethernet	-
#3	**syz_emit_ethernet**	100%
#4	**syz_emit_ethernet, syz_emit_ethernet$ipv4_igmp**	100%
#5	**syz_emit_ethernet**	100%
#6	**syz_emit_ethernet, syz_emit_ethernet$ipv4**	100%
#7	**recvmmsg, recvmsg**	100%
#8	**tee**	100%
#9	**getcwd**	100%
#10	**splice, tee**	100%
#11	**waitid**	100%
#12	**link**, linkat	50%
#13	**alarm**	100%
#14	**dup**	100%
#15	**mremap**	100%
#16	**mbind, set_mempolicy**	100%
#17	**unshare**	100%
#18	io_destroy	-
#19	mmap, **mprotect, munmap, wait4**	75%
#20	**getpeername, getsockname**, mmap, **mprotect**, munmap	60%
#21	**getsockopt**, mmap, **mprotect**, munmap	50%
#22	**recvmmsg**	100%
#23	**sendmmsg**	100%
#24	**io_setup**, mmap, **mprotect**, munmap	50%
#25	sendfile, **syz_emit_ethernet$ipv6**	50%
#26	**getsockopt**, syz_emit_ethernet	50%

TABLE 10: The inferred syscalls for the patch targets.

Patch ID	Inferred Syscalls	Precision
#1	**getxattr, lgetxattr**	100%
#2	**lremovexattr, removexattr**	100%
#3	**syz_emit_ethernet, syz_emit_ethernet$arp**	100%
#4	**syz_emit_ethernet, syz_emit_ethernet$ipv6, syz_emit_ethernet$ipv6_icmp**	100%
#5	**socketpair**	100%
#6	**syz_emit_ethernet, syz_emit_ethernet$ipv4**	100%
#7	**syz_emit_ethernet, syz_emit_ethernet$ipv4**	100%
#8	**syz_emit_ethernet, syz_emit_ethernet$ipv4**	100%
#9	**fstatfs, openat$ptmx**, statfs, syz_open_pts	75%
#10	**fstatfs, pipe, pipe2**, statfs	75%
#11	pread64, preadv, preadv2, read, readv	-
#12	**syz_emit_ethernet, syz_emit_ethernet$ipv6, syz_emit_ethernet$ipv6_icmp**	100%
#13	**connect**, syz_emit_ethernet$ipv4_udp, syz_emit_ethernet$ipv6_udp	33%
#14	**setsockopt**	100%
#15	**syz_emit_ethernet, syz_emit_ethernet$ipv4, syz_emit_ethernet$ipv4_igmp**	100%
#16	**splice**	100%
#17	**splice**	100%
#18	**sendfile**	100%
#19	**syz_emit_ethernet**	100%
#20	**syz_emit_ethernet, syz_emit_ethernet$ipv6**	100%
#21	sendfile, **syz_emit_ethernet**	50%
#22	sendfile, syz_emit_ethernet$ipv4_icmp, syz_emit_ethernet$ipv6_icmp	-

itives in the inference results. Thus, as we describe in §3.4, G-Fuzz adjusts the selection probability of each syscall dynamically according to the current fuzzing state. To verify the effectiveness of this strategy, we record the selection probability of each inferred syscall during the fuzzing process. Fig. 10 presents the curve of the selection probability of each inferred syscall of patch target #9 with the fuzzing time increasing. For this target, G-Fuzz infers four target related syscalls: `fstatfs`, `statfs`, `openat$ptmx` and `syz_open_pts`. From Fig. 10, we can observe that the initial selection probability for the four inferred syscalls are equal. With the fuzzing going on, G-Fuzz finds that using `fstatfs` can reduce the distance to the target effectively. Thus, the selection probability of `fstatfs` increases, while the probabilities of the rest three syscalls decrease. At time 1,354 seconds, G-Fuzz successfully triggers the target, which is denoted in the red line of the figure, and the PoC contains the syscall `fstatfs`. This observation indicates that the proposed selection probability scheduling strategy can mitigate the impacts caused by the false positives of the inference.

Ablation Study. We conduct ablation study on the target related syscall inference method. we compare G-Fuzz to G-Fuzz-noInfer (G-Fuzz that does not leverage the inferred syscalls) and Syzkaller. Fig. 11 shows the experimental results, we can obverse that G-Fuzz uses significantly less time in triggering the targets than G-Fuzz-noInfer and Syzkaller. G-Fuzz outperforms G-Fuzz-noInfer and Syzkaller in 10 targets. G-Fuzz-noInfer outperforms Syzkaller in 8 targets (except for target 2 and 10). The results can prove the effectiveness of the target inference optimization of G-Fuzz.
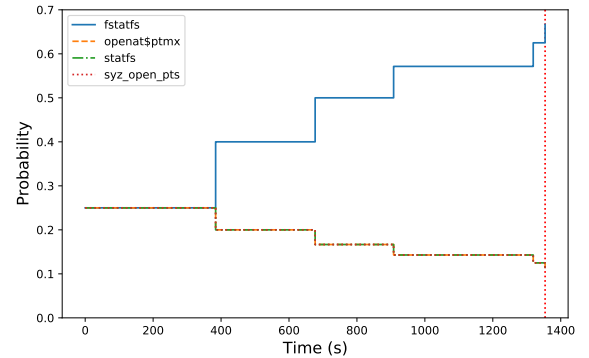

Fig. 10: The change of the selection probability of each inferred syscall during the fuzzing process.

6.3 Exploration and Exploitation Dynamic Switch

The Selection of Parameters. To test the impacts of the selection of parameters (i.e., T_a and T_b), we conduct the experiments. Specifically, we select 10 targets to test the performance of G-Fuzz with different parameter settings. Each experiment is conducted in five repetitions. As the value space of the two parameters (T_a, T_b) is very large, for convenience, we set $T_a = T_b$ in the experiments. Table 11 shows the experimental results. The second column represents the experimental results of G-Fuzz with the setting in this paper ($T_a = 5min, T_b = 10min$). From the table, we can observe that in most cases, G-Fuzz has the best performance with the paper's setting. On two targets, G-Fuzz with the
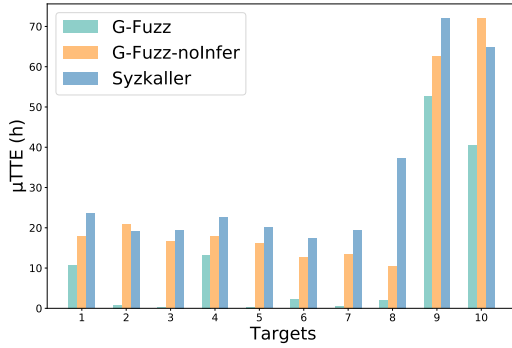
Fig. 11: Ablation study 2: target related syscall inference optimization.

setting ($T_a = T_b = 1min$) has the best performance. If the value of T_a and T_b is set too large, it may hinder the performance of G-Fuzz.

Ablation Study. We conduct ablation study on the exploration and exploitation dynamic method. we compare G-Fuzz to G-Fuzz-only-explore (G-Fuzz that only uses exploration state), G-Fuzz-only-exploit (G-Fuzz that only uses exploitation state) and Syzkaller. Fig. 12 shows the experimental results. We can find that G-Fuzz has the best performance, which proves the effectiveness of the exploration and exploitation dynamic switch optimization. As G-Fuzz-only-explore has better performance than G-Fuzz-only-exploit in 4 out of 10 targets (target 2, 3, 8, 10), it is hard to determine which is better. In total, G-Fuzz-only-explore and G-Fuzz-only-exploit use less time in triggering targets than Syzkaller, which also demonstrate the powers of the other optimizations of G-Fuzz in directed fuzzing.
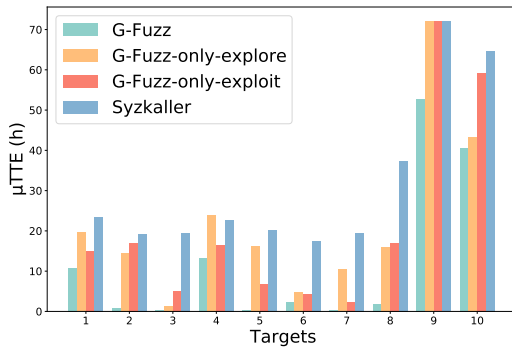

Fig. 12: Ablation study 3: exploration and exploitation dynamic switch optimization.

6.4 The Deployment and the Application of G-Fuzz

We have applied G-Fuzz in directly fuzzing the customized gVisor, called gVisor-x, which is widely deployed in Ant Group, a world-leading IT company. We select several vulnerable code from Syzbot [35], and if the vulnerable code lies in gVisor-x, it will be considered as the target to conduct directed fuzzing test. Leveraging G-Fuzz, we successfully discovered seven new vulnerabilities in gVisor-x, which the developers of Ant Group have confirmed. For commercial

reasons, we present partial information of the discovered new vulnerabilities in Table 12. The found vulnerabilities may have severe impacts on the security and reliability of the business service. For instance, a malicious process inside the container can make the gVisor panic, so the business process will also be killed, causing Deny of Service. Moreover, the race condition vulnerability may potentially allow the attacker to gain privilege.

6.5 The Generalisability of G-Fuzz

Despite we focus on solving the directed fuzzing problem for gVisor in this paper. The three core methods of G-Fuzz are general and scalable, which address the critical challenges for applying directed fuzzing on other OS kernels. First, the static distance calculation of G-Fuzz can be applied to any program that can be compiled into LLVM IR. Second, most of the inference rules for identifying the target related syscalls are suitable for other OS kernels. Third, the exploration and exploitable dynamic switch method can be applied to directed fuzzing for any program. Moreover, to prove the scalability of G-Fuzz, we implement a simple directed fuzzing prototype for Linux kernels called G-Fuzz-Linux, which is based on the methods of G-Fuzz. We conduct experiments to compare the performance of G-Fuzz-Linux with Syzkaller in directed fuzzing Linux kernels. Here we select ten Linux kernel bugs as the targets and repeat each experiment in five times. Table 13 presents the experimental results. We can observe that G-Fuzz-Linux has better performance than Syzkaller in reproducing Linux kernel bugs. Bug 3, 4, and 10 can only be triggered by G-Fuzz-Linux. Moreover, we open source the code of G-Fuzz-Linux [11] to facilitate the future research.

7 DISCUSSION: THREATS TO VALIDITY

Although G-Fuzz has good performance in directed fuzzing for gVisor, it can be further improved in the following ways.

Optimization of the Inference Methods. The proposed target related syscall inference method creatively reduces the huge input space for kernel directed fuzzing. The experimental results also prove the effectiveness of this method. The existing inference methods can be improved from the following ways. First, the existing methods are based on the expert knowledge. Some inference rules are easily to generate by analyzing the target code and the PoC input, while some rules still need much expert knowledge. In essence, the inference rules reflect the relations of the properties of the target code and the PoC inputs. Thus, it is feasible to develop an automatic inference rule generation method from large-scale corpus with machine learning algorithms in the future. Second, the inference rules are generated by static analysis. In the future, we will study how to generate the inference rules dynamically during the fuzzing process.

Discovery of More Vulnerabilities. Due to the limitation of the targets and the fuzzing time, we do not find a large amount of new vulnerabilities in the current paper. As G-Fuzz has outstanding performance in directed fuzzing, which is also demonstrated from the existing experimental results, we are confident that G-Fuzz can discover new vulnerabilities when testing more targets.

TABLE 11: The performance of G-Fuzz with different T_a and T_b settings.

Targets	G-Fuzz						Syzkaller
	paper-setting	$T_a = T_b$=1min	$T_a = T_b$=30min	$T_a = T_b$=1h	$T_a = T_b$=4h	$T_a = T_b$=8h	
1	**10.81**	18.79	16.86	11.28	13.32	15.03	23.52
2	**0.80**	9.90	15.85	7.03	12.00	8.73	19.17
3	0.23	**0.14**	9.70	0.22	4.98	0.16	19.39
4	13.28	17.36	17.49	20.65	22.71	18.10	22.62
5	**0.35**	5.24	1.59	5.29	3.13	12.01	20.11
6	2.21	**0.09**	0.15	1.35	5.32	5.30	17.41
7	**0.42**	0.71	11.14	0.95	0.85	5.53	19.50
8	**1.90**	4.57	4.91	5.31	72.00	72.00	37.29
9	**52.77**	54.37	57.49	61.60	72.00	72.00	72.00
10	**40.57**	23.30	30.18	29.27	72.00	72.00	64.74

TABLE 12: The new vulnerabilities of gVisor-x found by G-Fuzz.

Type	Description
buffer overflow	panic: runtime error: slice bounds out of range [12:LINE]
buffer overflow	panic: runtime error: index out of range [260] with length 260
stack overflow	fatal error: heapBitsSetType: called with non-pointer type
logic error	panic: Unknown syscall NUM error: rename across inodes with different implementations
floating point error	panic: runtime error: floating point error
concurrency bug	fatal error: s.freeindex >s.nelems
concurrency bug	panic: gofer.dentry.decRefNoCaching() called without holding a reference

TABLE 13: The performance of G-Fuzz-Linux and Syzkaller in Linux kernel bug reproduction.

Bug	Fuzzer	Runs	μTTE (h)	Version
1	G-Fuzz-Linux	5	4.06	
	Syzkaller	1	65.52	
2	G-Fuzz-Linux	5	4.22	4.19.204
	Syzkaller	1	71.17	
4	G-Fuzz-Linux	3	60.79	
	Syzkaller	0	72.00	
3	G-Fuzz-Linux	1	61.40	
	Syzkaller	0	72.00	4.19.199
6	G-Fuzz-Linux	5	18.27	
	Syzkaller	1	70.91	
7	G-Fuzz-Linux	5	36.71	4.19.194
	Syzkaller	1	68.49	
5	G-Fuzz-Linux	5	38.59	
	Syzkaller	1	58.64	
8	G-Fuzz-Linux	5	40.77	4.19.186
	Syzkaller	1	70.89	
9	G-Fuzz-Linux	5	6.39	
	Syzkaller	1	59.16	
10	G-Fuzz-Linux	3	37.92	4.19.180
	Syzkaller	0	72.00	

8 RELATED WORK

Kernel Fuzzing. Fuzzing is a popular research topic in software and system security [33], [36]–[38]. There are many research works focus on kernel fuzzing [30], [39]–[47]. IMF [39] is a model-based API fuzzer that targets macOS. Syzkaller [48] is a widely used kernel fuzzer and can be used for multiple OS kernels such as Linux, OpenBSD, Windows, gVisor, etc. kAFL [40] proposes a hardware-assisted method to record coverage with low overhead, and can be applied to many OS kernels. Moonshine [30] leverages static analysis to extract the dependencies across different syscalls to further distilling the initial seeds for OS kernel fuzzing. In G-Fuzz, we also analyze the dependencies of the syscalls to generate semantically correct seeds. Some OS kernel fuzzers aim at drivers [41]–[43]. DIFUZE [42] is an interface aware fuzzer for Android device drivers. Moreover, some kernel fuzzing works aim at testing the file system of the kernel [44], [45] or detecting some specific vulnerabilities [46], [47].

Static Analysis for Kernel. A plethora of research has adopted static analysis to discover different vulnerabilities of the OS kernels [49]–[55]. K-Miner [49] is a framework for detecting memory corruption related vulnerabilities of the Linux kernel. Deadline [50], Dftinker [51] and [52] focus on discovering double-fetch vulnerabilities in the Linux kernel. DCNS [53] aims at detecting non-sleep defects in the Linux kernel. For missing security checks, LRSan [54] and Crix [55] are proposed for detecting such vulnerabilities. The kernel code is complex and has many indirect calls, which brings challenges to the static analysis. Lu et al. [22] propose multi-layer type analysis to identify the indirect calls with outstanding performance, demonstrating the effectiveness of type analysis methods. In this paper, we also leverage type analysis to find the indirect calls of gVisor.

9 CONCLUSION

We propose G-Fuzz, a directed fuzzing framework for gVisor, which incorporates three core methods including *lightweight and fine-grained distance calculation*, *target related syscalls inference and utilization*, and *exploration and exploitation dynamic switch*. The methods of G-Fuzz are general and scalable, which can be applied on other OS kernels like Linux kernels. Compared to the state-of-the-art kernel fuzzers, G-Fuzz achieves much more efficient and stable performance. Moreover, G-Fuzz has been deployed in the industry for the customized gVisor, and has successfully discovered multiple real-world vulnerabilities.

ACKNOWLEDGMENTS

REFERENCES

[1] "Stripe," https://stripe.com.
[2] "Digitalocean," https://www.digitalocean.com.
[3] "Cloudflare," https://www.cloudflare.com.
[4] Google, "gVisor," https://github.com/google/gvisor.
[5] P. D. Marinescu and C. Cadar, "Katch: High-coverage testing of software patches," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (FSE)*, 2013, pp. 235–245.
[6] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 474–484.
[7] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2329–2344.
[8] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: towards a desired directed grey-box fuzzer," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 2095–2108.
[9] "Dijkstra's algorithm," https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.
[10] "G-fuzz: A directed fuzzing framework for gvisor," https://github.com/zjuchenyuan/gfuzz.
[11] "G-fuzz-linux," https://github.com/zjuchenyuan/G-Fuzz-Linux.
[12] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," *Acm Sigplan Notices*, vol. 46, no. 6, pp. 504–515, 2011.
[13] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009, pp. 474–484.
[14] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *International Static Analysis Symposium*. Springer, 2011, pp. 95–111.
[15] J. Ye, R. Li, and B. Zhang, "Rdfuzz: Accelerating directed fuzzing with intertwined schedule and optimized mutation," *Mathematical Problems in Engineering*, vol. 2020, 2020.
[16] W. Wang, H. Sun, and Q. Zeng, "Seededfuzz: Selecting and generating seeds for directed fuzzing," in *Proceedings of the 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, 2016, pp. 49–56.
[17] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020, pp. 2255–2269.
[18] G. Lee, W. Shim, and B. Lee, "Constraint-guided directed greybox fuzzing," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021, pp. 3559–3576.
[19] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "SemFuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2139–2154.
[20] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, "Binary-level directed fuzzing for use-after-free vulnerabilities," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 47–62.
[21] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, Citeseer, 1994.
[22] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 1867–1881.
[23] S. Biallas, M. C. Olesen, F. Cassez, and R. Huuck, "Ptrtracker: Pragmatic pointer analysis," in *Proceedings of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2013, pp. 69–73.
[24] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi, "On the effectiveness of type-based control flow integrity," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 28–39.
[25] "Rapid Type Analysis (RTA) for Go," https://pkg.go.dev/golang.org/x/tools/go/callgraph/rta.
[26] O. Fabry, "go-callvis," https://github.com/ofabry/go-callvis.
[27] "Syscall descriptions," https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions.md.
[28] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Fuzzing error handling code using context-sensitive software fault injection," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020, pp. 2595–2612.
[29] Q. Wu, A. Pakki, N. Emamdoost, S. McCamant, and K. Lu, "Understanding and detecting disordered error handling with precise function pairing," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021.
[30] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing OS fuzzer seed selection with trace distillation," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018, pp. 729–743.
[31] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "EcoFuzz: Adaptive Energy-Saving greybox fuzzing as a variant of the adversarial Multi-Armed bandit," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020, pp. 2307–2324.
[32] Google, "GoLLVM," https://go.googlesource.com/gollvm/.
[33] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, and T. Wang, "UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzer," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021.
[34] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.
[35] "syz-bot: gVisor," https://syzkaller.appspot.com/gvisor.
[36] C. Lyu, S. Ji, C. Zhang, Y. Li, W. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019, pp. 1949–1966.
[37] X. Zhang, H. Liang, Y. Xia, Y. Pu, and S. Ji, "Parallel fuzzing: a survey," *Information Countermeasure Technology*, vol. 1, no. 1, pp. 24–42, 2022.
[38] Y. Li, S. Ji, C. Lyu, Y. Chen, J. Chen, Q. Gu, C. Wu, and R. Beyah, "V-fuzz: Vulnerability prediction assisted evolutionary fuzzing for binary programs," *IEEE Transactions on Cybernetics*, vol. 52, no. 5, pp. 3745–3756, 2022.
[39] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2345–2358.
[40] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017, pp. 167–182.
[41] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in *Network and Distributed System Security (NDSS)*, 2019.
[42] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2123–2138.
[43] H. Peng and M. Payer, "Usbfuzz: A framework for fuzzing USB drivers by device emulation," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020, pp. 2559–2575.
[44] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 818–834.
[45] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding bugs in file systems with an extensible fuzzing framework," *ACM Transactions on Storage (TOS)*, vol. 16, no. 2, pp. 1–35, 2020.
[46] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 754–768.
[47] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1643–1660.
[48] Google, "Syzkaller," https://github.com/google/syzkaller.
[49] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-miner: Uncovering memory corruption in linux." in *NDSS*, 2018.
[50] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in os kernels," in *Proceedings of the*

39th IEEE Symposium on Security and Privacy (SP). IEEE, 2018, pp. 661–678.

[51] Y. Luo, P. Wang, X. Zhou, and K. Lu, "Dftinker: Detecting and fixing double-fetch bugs in an automated way," in *International Conference on Wireless Algorithms, Systems, and Applications*. Springer, 2018, pp. 780–785.

[52] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017, pp. 1–16.

[53] J.-J. Bai, J. Lawall, W. Tan, and S.-M. Hu, "Dcns: automated detection of conservative non-sleep defects in the linux kernel," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 287–299.

[54] W. Wang, K. Lu, and P.-C. Yew, "Check it again: Detecting lacking-recheck bugs in os kernels," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1899–1913.

[55] K. Lu, A. Pakki, and Q. Wu, "Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019, pp. 1769–1786.