SyzDirect: Directed Greybox Fuzzing for Linux Kernel

Xin Tan*
Fudan University
18212010028@fudan.edu.cn

Yuan Zhang*
Fudan University
yuanxzhang@fudan.edu.cn

Jiadong Lu
Fudan University
21210240091@m.fudan.edu.cn

Xin Xiong
Fudan University
xiongx18@fudan.edu.cn

Zhuang Liu
Fudan University
20210240039@fudan.edu.cn

Min Yang
Fudan University
m_yang@fudan.edu.cn

ABSTRACT

Bug reports and patch commits are dramatically increasing for OS kernels, incentivizing a critical need for kernel-level bug reproduction and patch testing. Directed greybox fuzzing (DGF), aiming to stress-test a specific part of code, is a promising approach for bug reproduction and patch testing. However, the existing DGF methods exclusively target user-space applications, presenting intrinsic limitations in handling OS kernels. In particular, these methods cannot pinpoint the appropriate system calls and the needed syscall parameter values to reach the target location, resulting in low efficiency and waste of resources.

In this paper, we present SyzDirect, a DGF solution for the Linux kernel. With a novel, scalable static analysis of the Linux kernel, SyzDirect identifies valuable information such as correct system calls and conditions on their arguments to reach the target location. During fuzzing, SyzDirect utilizes the static analysis results to guide the generation and mutation of test cases, followed by leveraging distance-based feedback for seed prioritization and power scheduling. We evaluated SyzDirect on upstream Linux kernels for bug reproduction and patch testing. The results show that SyzDirect can reproduce 320% more bugs and reach 25.6% more target patches than generic kernel fuzzers. It also improves the speed of bug reproduction and patch reaching by a factor of 154.3 and 680.9, respectively.

CCS CONCEPTS

• **Security and privacy → Software and application security; Operating systems security**.

KEYWORDS

Directed greybox fuzzing; Kernel fuzzing; OS security; Static analysis

ACM Reference Format:
Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. 2023. SyzDirect: Directed Greybox Fuzzing for Linux Kernel . In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623146

*co-first authors

1 INTRODUCTION

Kernel is the most important software component which supports the entire operating system (OS) and all user applications. Thus, the security of OS kernels is crucial and has gained massive attention in recent years. Numerous static analysis techniques [8, 13, 16, 27, 29, 30, 41, 46] and fuzzing tools [4, 18, 24, 38, 40, 42] have emerged and discovered a large number of security vulnerabilities in, for instance, the Linux kernel. This incurs tremendous pressure on kernel developers to analyze the bugs and develop high-quality patches. In such a circumstance, directed greybox fuzzing (DGF)—aiming to prioritize fuzz testing on a specific code location [10, 14, 15, 22, 26, 47]—is a promising technique for help. ❶ DGF can automate the reproduction of bugs if their reports do not come with a proof-of-concept (PoC) input. ❷ DGF can stress-test the developed patches to assess their quality, reducing manual review efforts. However, the existing DGF techniques primarily focus on user-space applications [10, 14, 15, 22, 26, 47], which cannot be adapted for OS kernels easily. The state-of-the-art DGF technique for OS kernels is GREBE [28]. However, GREBE is a method tailored to a specific task (i.e., exploring more error behaviors of a kernel bug.), rather than a generic directed fuzzing solution. It requires the PoC program as its input, which is not available in typical DGF scenarios.

In general, the existing DGF techniques employ two major strategies to approach the target location quickly: *distance-guided exploration* [10, 14, 15, 26] and *invalid input pruning* [22, 47]. The first strategy gathers runtime feedback to calculate the distance to reach the target location and prioritizes test cases with a shorter distance. The second strategy filters out test cases that cannot reach the target location or follow an infeasible execution path, further boosting the exploration.

While the generic strategies above can be applied to OS kernels, achieving effective kernel-level DGF faces a series of obstacles that the available tools do not consider or overcome. (1) Unlike user-space applications which take a fixed entry point, OS kernels implement hundreds of system calls (abbreviated as syscalls) as the main interface to interact with the kernel. For instance, Linux kernel has around 330 primitive syscalls and the SOTA kernel fuzzer, Syzkaller [18], defines nearly 4,200 syscall variants for fuzzing. Thus identifying the correct entry syscall and variant to reach the target site is important for kernel-level DGF to avoid wasting time exploring the infeasible syscalls. However, applying existing control flow analysis techniques to identify entry syscall introduces a lot of false positives due to the heavy use of indirect calls in Linux kernel. In addition, there's no work that considers how to match the kernel code to the syscall variants. (2) In addition to syscalls, it is also important to prepare needed syscall arguments to meet the constraints on the execution path to the target. Otherwise, DGF would easily get stuck in exploring the argument space. However, existing generic static analysis methods, like the precondition analysis adopted by Beacon [22], can unlikely work well due to the complexity of the kernel. On the one hand, tracing the execution of syscall requires analyzing the deep code. On the other hand, the kernel code involves a great many indirect calls, linked list operations, nested data structures, and many layers of pointer dereference, making accurate data flow analysis of deep codes challenging.

To address the above challenges, we present SyzDirect, the first general directed greybox fuzzing solution for Linux kernel. SyzDirect takes a target code location in the target kernel version as input and aims to stress-test the target code. SyzDirect leverages novel static analysis, which exploits the design of Linux kernel and Syzlang [19], to identify important information to reach the target site. First, SyzDirect identifies syscall variants that serve as the entry point to the target site. To achieve this, SyzDirect matches kernel functions with syscall variants by modeling the operations they perform and the resources they operate on, which is challenging for traditional static analysis. Second, SyzDirect infers the syscall variants that are dependent on the entry syscalls which set important contexts (e.g., generating suitable resources) for triggering the target. Third, SyzDirect leverages a light-weight method to identify the conditions on syscall arguments to reach the target and refine the argument description for entry syscalls. The method exploits the rich information of Syzlang descriptions to avoid heavy-weight data flow analysis. Then SyzDirect assembles the information as templates to guide the directed fuzzing. In particular, SyzDirect adopts a customized mutation algorithm that prefers to generate test cases following the templates and leverages distance-based feedback for seed prioritization and power scheduling.

We have implemented a prototype of SyzDirect based on Syzkaller [18] and LLVM [25]. We evaluate SyzDirect against Syzkaller and SyzGo (a variant of AFLGo [10] for Linux kernel) in two typical application scenarios of DGF: patch testing and bug reproduction. In addition, we have adapted GREBE[28] as a general DGF approach and compared its ability of reproducing bugs with SyzDirect. The results show that SyzDirect outperforms Syzkaller, SyzGo, and GREBE in terms of efficiency and effectiveness. For bug reproduction, SyzDirect could reproduce 320%, 281%, and 121% more bugs than Syzkaller, SyzGo, and GREBE, respectively. For bugs that Syzkaller and SyzGo could reproduce, SyzDirect reproduced 154.3x faster than Syzkaller and 81.9x faster than SyzGo. For patch testing, SyzDirect can cover 25.6% and 36.1% more targets than Syzkaller and SyzGo. Most of all, SyzDirect discovered 4 known insecure patches that other fuzzers could not. In addition, the experiment individually evaluated the benefits of each key component of SyzDirect for kernel-level DGF.

In summary, we make the following contributions:

- We propose a static analysis method to identify important information for kernel-level DGF, including the correct entry syscalls and the conditions on the arguments to reach the target.
- We present the design and implementation of SyzDirect, a directed greybox fuzzing framework for Linux kernel, which leverages the identified information to guide fuzzing.
- We conduct a comprehensive evaluation of SyzDirect against the generic kernel fuzzer and the existing DGF techniques. The results show that SyzDirect significantly outperforms them in both efficiency and effectiveness.

Available Artifact. We open-source SyzDirect under the Apache-2.0 license at https://github.com/seclab-fudan/SyzDirect. This artifact contains the prototype of SyzDirect as well as a modified Clang compiler for customized instrumentation.

2 BACKGROUND AND MOTIVATION

2.1 Directed Greybox Fuzzing

Distance-guided Exploration. Directed greybox fuzzing (DGF) aims to intensively test a target location in a program. Compared to generic greybox fuzzing, DGF prioritizes the exploration toward the target location and, thus, can approach the target location faster. Existing methods of DGF [10, 14, 26] typically leverage a certain type of distance to guide the fuzzing progress. They first measure the distance of each input to the target location and then assign more energy to mutate inputs closer to the target location. When designing distance metrics, AFLGo [10] and Hawkeye [14] consider the control-flow information, and CAFL [26] further incorporates the data-flow information.

Optimization via Input Pruning. Distance-based guidance is helpful but fails to accurately determine which inputs can hit the target location. Thus, it still allows time to be spent on test cases not reaching the target, leading to a waste of resources [22, 47]. To address this problem, recent research [22, 47] proposes further optimizing DGF by pruning inputs that cannot reach the target. Specifically, FuzzGuard [47] develops a deep-learning-based approach to predict the reachability of inputs and filter out the non-reachable ones before sending them to fuzzing. In contrast, Beacon [22] tailors static analysis to compute preconditions for reaching the target and early terminate inputs following execution paths not satisfying the preconditions.

2.2 DGF for OS Kernels

The existing DGF techniques primarily focus on user-space applications. However, OS kernels also represent an important kind of targets where DGF is critically needed. Consider Linux kernel as an example. The Bugzilla [5] receives a mass of bug reports without a PoC, making reproduction and analysis extremely hard. DGF can help derive the missing PoCs to offload the burden of post-mortem analysis. Further, faulty patches are often committed to the Linux kernel [1, 2, 43], which can also be identified and avoided via DGF.

While the generic idea of DGF can be adapted to OS kernels, achieving effective kernel-level DGF face a series of obstacles that the existing methods cannot overcome.

Challenge I: Entry Points. Unlike user-space applications, OS kernels have many entry points represented as system calls (or

```
1   /* File: net/rds/send.c */
2   int rds_rm_size(...){
3         ...
4         //cmsg comes from the second argument of sendmsg
5         switch (cmsg->cmsg_type){
6             case RDS_CMSG_RDMA_ARGS:
7                 ...
8                 retval =
                 rds_rdma_extra_size(CMSG_DATA(cmsg), iov);
9         ...
10  }
11  /* File: net/rds/rdma.c */
12  int rds_rdma_extra_size(struct rds_rdma_args *args,
13        struct rds_iov_vector *iov){
14        ...
15        if (args->nr_local == 0)
16                return -EINVAL;
17        //lead to a warning when args->nr_local is large
18  +     iov->iov = kcalloc(args->nr_local,
19  +             sizeof(struct rds_iovec),
20  +             GFP_KERNEL);
21        ...
22  }
```

Figure 1: An exemplary faulty patch in Linux kernel [3]. The patch does not check the user-provided `args→nr_local`, which can lead to an excessive allocation size for `kcalloc` and a kernel warning.

syscalls). A target location is usually only reachable from a small subset of syscalls. For instance, the Linux kernel has around 330 syscalls while the faulty patch in Figure 1 can only be reached via `sendmsg` and `sendmmsg`. Worse yet, each native syscall is often a wrapper of many independent variants, and only certain variants can reach the target. For example, Syzkaller defines 468 variants for `sendmsg`, depending on which protocol/sub-protocol is used [6]. The faulty patch in Figure 1 is only reachable when `sendmsg` uses the Reliable Datagram Socket (RDS) protocol. In total, the Linux kernel has nearly 4,200 syscall variants based on Syzkaller's definition.

The first challenge of kernel-level DGF is identifying the correct syscalls and variants. Otherwise, we will waste tremendous time exploring the right entry point. However, the existing DGF methods neither consider this challenge nor provide solutions. Intuition suggests we may find the entry point via control flow analysis. This is not true. The faulty patch in Figure 1 is only reachable through the call chain of `sendmsg→...→rds_sendmsg→...→rds_rdma_extra_ size`, where `rds_sendmsg` is invoked via an indirect call and the pointer pointing to `rds_sendmsg` is set up in another syscall (`socket`). The only scalable method to resolve this indirect call is type-based pointer analysis, which, unfortunately, introduces false positives. For example, the state-of-the-art type-based pointer analysis [31] reports that 235 different syscalls can reach the faulty patch. Further, even given the primary syscall, control flow analysis cannot separate different variants as all variants share the same entry point (i.e., the primary system call).

Challenge II: Argument Preparation. Finding the correct entry point helps but remains insufficient for kernel-level DGF. To reach a target location, we must also prepare the needed arguments for the syscalls. Otherwise, DGF can easily get stuck in exploring the argument space.

Determining the needed arguments requires understanding the conditions enforced on them. Generic methods, like the precondition analysis employed by Beacon [22], can unlikely work. The conditions are often embedded in deep code. To associate them with the arguments, precondition analysis must accurately reason the long, complex trail from the deep code to the syscall entry. Consider the example shown in Figure 1. To reach the faulty patch, a field deeply nested in the second argument of `sendmsg` must satisfy a condition right before the function call to `rds_rdma_extra_size` (line 6 in Figure 1). The propagation of the argument field from the syscall entry to the condition site involves indirect calls, linked list operations, nested data structures, and many layers of pointer dereference, making accurate precondition analysis extremely challenging.

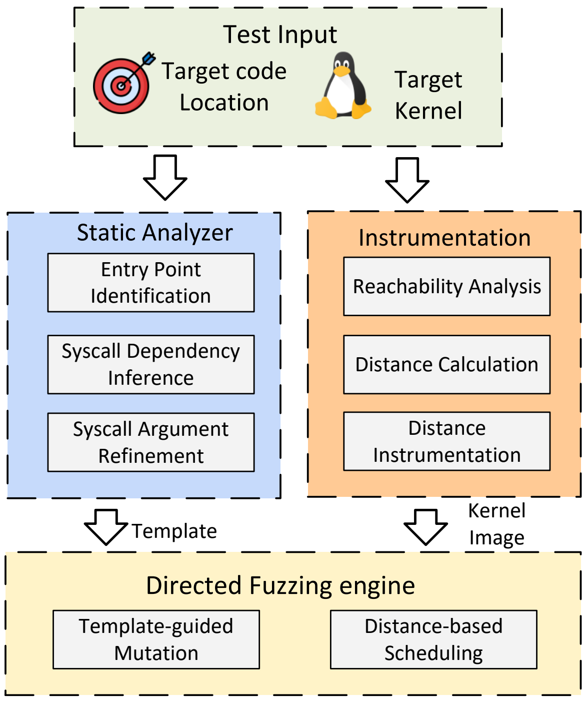

Figure 2: Overview of SyzDirect.

3 APPROACH OVERVIEW

In this paper, we propose SyzDirect, a general DGF solution for Linux kernel. SyzDirect takes a designated code location in the target kernel version as input and follows the workflow presented in Figure 2 to stress-test that location.

Entry Point Identification. Instead of relying on generic control flow analysis, we exploit the design of Linux kernel to identify the entry point. In essence, Linux kernel is a manager of resources abstracted as files, sockets, devices, etc. Syscalls are interfaces to operate on the resources (create, update, recycle, etc.), while kernel functions provide the actual implementations for the operations. This brings us the following key insight: *we can match kernel* *functions with syscalls based on **what** resources they operate on and **how** they operate on the resource.*

Consider the case shown in Figure 1 as an example. Based on information in the sound call graph (constructed with over-approximating point-to-analysis to resolve indirect calls), the target function `rds_rdma_extra_size` can only be reached via kernel function `rds_sendmsg`, which is a member function of data structure `rds_proto_ops`. Leveraging a model we build to describe the creation and operation of resources in the Linux kernel (see §4.1), we can figure out that `rds_proto_ops` is used by `rds_create` to register a socket with the `SOCK_SEQPACKET` type from the `AF_RDS` family. This way, we learn that `rds_rdma_extra_size` operates on a `AF_RDS SOCK_SEQPACKET` socket. By further referring to Syzkaller's descriptions (i.e., Syzlang) [6], we can see that only syscall variants from the `$rds` family (`socket$rds`, `bind$rds`, `sendmsg$rds`, etc.) provide interfaces to the same resource, enabling us to match our target function with those syscall variants. Finally, applying the matching result to prune the sound call graph, we can determine that the target function `rds_rdma_extra_size` is only reachable via `sendmsg$rds`.

Syscall Dependency Inference. Kernel-level DGF often requires a sequence of dependent syscalls to reach the target. For instance, reaching the faulty patch in Figure 1 mandates sequential execution of `socket$rds` and `sendmsg$rds` on the same socket.

We infer dependency between syscalls based on the resources they operate on, following the static learning method proposed by Healer [40]. Specifically, if syscall $\mathcal{A}$ may use resource created by syscall $\mathcal{B}$, we consider that $\mathcal{A}$ depends on $\mathcal{B}$. In the example shown in Figure 1, `socket$rds` creates the desired socket for `sendmsg$rds`, enabling us to establish their dependency. The remaining problem is to figure out *which* syscalls create/use *what* resources. This is a trivial task, as Syzlang has already offered such information at the syscall variant level.

Syscall Argument Refinement. Figuring out the syscall variants often automatically enforces some conditions on arguments. In Figure 1, knowing that `sendmsg$rds` is the desired variant enables us to restrict `sendmsg`'s first argument to be a `AF_RDS SOCK_SEQPACKET` socket descriptor. But this is not enough because many conditions are variant-independent (e.g., the condition at line 6 of Figure 1 checks the message type, which is not regulated by syscall variant).

To identify variant-independent conditions on arguments, we leverage the information available in Syzlang. Syzlang has described the conditions of syscall arguments according to the resources (e.g., a socket) or sub-resources (e.g., a socket for a specific protocol). For instance, `sendmsg$rds` supports seven message types (RDS_CMSG_RDMA_ARGS, RDS_CMSG_RDMA_DEST, RDS_CMSG_RDMA_MAP, etc.), and Syzlang specifies the conditions on the arguments (and their recursively nested fields) of `sendmsg` for each type. In our approach, we identify all conditions that dominate the target location or affect indirect calls, followed by an attempt to match each identified condition with the argument conditions included in Syzlang. The matching is based on lightweight methods like literal/value-based comparison, which, once successful, enables us to transfer a code condition to an argument condition without any data flow or precondition analysis. For instance, the condition at line 6 of Figure 1 involves a union value named "RDS_CMSG_RDMA_ARGS", which matches the argument conditions in Syzlang for the "RDS_CMSG_RDMA_ARGS" message type. Thus, we borrow the corresponding argument conditions from Syzlang for `sendmsg` during DGF.

Directed Kernel Fuzzing. We adapt Syzkaller [18] to perform directed fuzzing. ❶ Our analyses above identify the sequence of necessary syscalls and the conditions on their arguments to reach the target location, which we assemble as templates to guide Syzkaller. Specifically, we customize the seed mutation process to generate test cases following the templates with a dominating probability. This way, we reduce the testing further away from the target location. ❷ We force Syzkaller to prioritize the mutations of seeds closer to the target location and assign higher energy to those seeds. Doing so enables us to approach the target location at a faster pace. ❸ We extend the scheme to preserve test cases. Besides test cases covering new code edges, we also preserve those resulting in a shorter distance to the target location, considering that they represent a closer step to the target.

Summary. To facilitate directed kernel fuzzing, SyzDirect features two new techniques, i.e., entry point identification (§4.1) and syscall argument refinement (§4.3). By exploiting the unique design of the Linux kernel and leveraging the syscall descriptions written in Syzlang, these two techniques overcome the limitations of existing static analysis, i.e., identifying correct syscalls for triggering the target site and preparing the needed syscall arguments, as introduced in §2.2. Besides, SyzDirect infers syscall dependencies by adapting the static learning algorithm in Healer [40] (§4.2). To our knowledge, SyzDirect is the first solution to automatically identify the syscall variants (not only primitive syscalls) and parameters that are required to reach a specific target site in deep kernel code, with the help of all these static analysis techniques.

In addition, SyzDirect extends Syzkaller to support directed fuzzing from two aspects. On the one hand, SyzDirect introduces tailored seed mutation and generation strategies (§4.5) to cooperate with the templates generated by the static analysis. On the other hand, SyzDirect realizes distance calculation (§4.4) and distance-guided scheduling by adapting and optimizing the algorithms of AFLGo [10]. Combining the template-assisted mutation and distance guidance, SyzDirect can effectively approach given target locations in the kernel.

4 DESIGN

4.1 Entry Point Identification

Our goal is to identify syscall variants that can serve as the entry point to the target site (we denote them as entry syscalls). To overcome the obstacles of entry point identification introduced in §2.2, we propose a new approach which *matches kernel functions with syscall variants by modeling what resources they operate on and how they operate on the resource.* Our new approach comes from two observations about the Linux kernel. ❶ Each syscall variant describes a specific operation on a specific resource in Syzlang language, providing a more fine-grained representation of kernel functionality compared to the coarse-grained primitive syscalls. ❷ After entering a syscall entry, the kernel performs function dispatch to determine which function needs to be executed. The dispatch process parses the syscall arguments to determine what resources are being processed and what operation should be performed, and thus determines which function to execute subsequently. Based on these observations, we propose to match the syscall variants to the kernel functions by modeling the operations performed and the resources operated on in a uniform way. Assisted with the matching results, SyzDirect could locate the entry point at the syscall variant level and reduce false positives due to inaccurate indirect call analysis.

Anchor Functions. Considering that not all kernel functions explicitly reflect the resources they operate on and the operations they perform, we cannot model all functions in a straightforward way. Since the process of function distribution naturally involves the handling of resources and the determination of operations, We mainly analyze the dispatch process to understand what resources are manipulated and what operations are performed by the subsequent codes. We denote the first functions executed after the dispatch process as anchor functions. We first model and match the anchor functions to the syscall variants. For other functions, we identify the anchor functions from which this function can be reached and map it to the syscall variants corresponding to those anchor functions.

Operation Modeling. In general, the primitive syscall entry corresponds to a set of related operations and the kernel determines which specific operation to perform based on the value of command parameters passed in. We therefore take the syscall names as well as the value of the command parameters to model how the kernel function and Syzlang variants operate on resources respectively. For the syscall variant analysis, we extract the syscall name directly from the Syzlang description. We then parse all its parameters and use the numeric constants among them as the command value. For the kernel function analysis, we combine control flow and data flow analysis to extract the command values. In particular, given a syscall name, we perform forward analysis from its code entry and locate all switch statements from the CFG. We then perform data flow analysis to determine whether the variables of the switch statement are related to the syscall parameters. For the parameter-tainted switches, we take the functions in each case branch as anchor functions and use the constant value of that case as their command value.

```
// Command parameter (i.e., code) is set to KEYCTL_UPDATE
keyctl$update(code const[KEYCTL_UPDATE], ...)
// Its operation is modeled as [keyctl,KEYCTL_UPDATE]
```

(a) Operation Modeling for `keyctl$update`

```
   /* code snippet File: security/keys/keyctl.c*/
1  SYSCALL_DEFINE5(keyctl,...,option, ...){
3      //perform function dispatch based on the incoming command
(i.e., option)
4      switch(option){
5      ...
6      case KEYCTL_UPDATE:
7          return keyctl_update_key(...);//anchor function
8          //operation for keyctl_update_key is modeled as
[keyctl,KEYCTL_UPDATE]
9      ...}
```

(b) Operation Modeling for Anchor Function corresponding to `keyctl$update`

Figure 3: Example of Operation Modeling.

Figure 3 illustrates an example of operation modeling. The syscall variant `keyctl$update` takes `KEYCTL_UPDATE` as its command value to indicate that the variant performs the operation to update the key. We thus model its operations as [keyctl, KEYCTL_UPDATE]. From the code of `keyctl`, the switch statement takes the first parameter of the syscall as its condition and leads to several case branches. Therefore, we locate anchor functions for each case branch and extract the constant values from case statements to model the operation for each anchor function. For example, we extract `KEYCTL_UPDATE` from line 6 and model the anchor function `keyctl_update_key` in line 8 as [keyctl, KEYCTL_UPDATE].

Resource Modeling. As the same resource may be named and defined differently between the kernel code and the Syzlang description, we cannot model them with resource names. In order to build a resource model across Syzlang descriptions and kernel code, we propose to model them using invariants in resource creation. Specifically, in order to specify a resource type, creating a resource requires a corresponding string or numeric constants. For example, creating device and file system resources requires a constant string indicating the file system/device path while creating sockets requires the family and socket type. We therefore model a resource with all the string and numeric constants associated with the creation of this resource.

For the syscall variants, we can obtain the resource it takes by parsing its parameters. To model the obtained resource, we first locate the syscall variant that created the resource and then parse the parameters of the creation syscall variant, extracting all constant strings and constant values from it. In the Syzlang descriptions shown in Figure 4, `sendmsg$rds` takes a resource named `sock_rds` as input in line 2. The producer of `sock_rds` takes several constants as input in line 1, we thus model `sock_rds` with the list of constants [AF_RDS,SOCK_SEQPACKET]. The case in line 6 is similar, we model `fd_i915` as the constant string involved in its creation (i.e., `openat$i915`). In particular, we removed the path prefix from the string and kept only the name of the device.

```
1  socket$rds(domain const[AF_RDS], type
const[SOCK_SEQPACKET], ...) sock_rds
2  sendmsg$rds(fd sock_rds, ...)
3  //sock_rds is modeled as [AF_RDS,SOCK_SEQPACKET]
4
5  openat$i915(fd const[AT_FDCWD], file ptr[in, string["/dev/
i915"]], ...) fd_i915
6  ioctl$DRM_IOCTL_I915_GETPARAM(fd fd_i915, ...)
7  //fd_i915 is modeled as ["i915"]
```

Figure 4: Example of Resource Modeling for Syscall Variants.

For kernel functions, we determine the required resources by analyzing the target of the indirect call. We observe that different types of resources are usually assigned a unique virtual table-like data structure in the kernel. After entering the syscall entry, the kernel performs function dispatch via an indirect call. This indirect call queries the virtual table corresponding to this resource, thus directing the control flow to the code that performs the corresponding functionality. Therefore, we take all the functions in the virtual table as anchor functions and they all belong to the resource which the virtual table belongs to. However, there is still a gap between the virtual table of a resource and the invariants in the creation of this resource.

To resolve this gap, we investigated the resource creation process and observed that most of the resource creation process in the kernel follows such a manner. ❶ During module loading or kernel initialization, the kernel invokes some generic **registration functions** which take key information, such as constant information about the resource and the resource **creation function** as arguments. ❷ When creating a resource, the kernel invokes the corresponding creation function. ❸ The resource creation function will assign other constants and virtual tables to the kernel structure which represents the resource object. Based on these observations, we develop a customized static analysis to correlate the virtual tables to the invariants of resources (i.e., constant string and numeric values). Our approach consists of three steps. First, we manually collect a list of commonly used registration functions and locate all their callsites. For each call site, we exploit the domain knowledge to extract key constant values and the **creation function** based on the variables' names. Second, we look into the **creation function**. We analyze all of the assignment statements and identify the assignment of resource-related constants and the assignment of resource virtual tables based on variable names and variable types. Third, for each callsite of the registration function, we take all the extracted resource-related constants to model the resources corresponding to the virtual table. Our approach requires some domain knowledge such as registration functions and the names of the key variables and key structures related to resource creation. In total, we have manually collected about 20 registration functions and key variables involving general resources such as sockets, devices and filesystems, covering most of the syscall variants.

Take the motivating case in Figure 1 as an example. Triggering `rds_rdma_extra_size` involves an indirect call, which directs the control flow to `rds_sendmsg`. Thus we denote `rds_sendmsg` as an anchor function and inspected all its references. It turns out that `rds_sendmsg` belongs to `rds_proto_ops` which is a virtual table-like structure of RDS socket. Next, we analyze the resource creation to determine which resource-related constants the `rds_proto_ops` correspond to. The creation of RDS socket is shown in Figure 5. First, the initialization of RDS module invokes a socket registration function in line 4. By analyzing its argument and its recursively nested fields, we obtain the family type and the create function We then further trace into `rds_create` and obtain the assignment of `rds_proto_ops` in line 22. What's more, we could find a check on the socket type in line 16 and informs the value of socket type is `SOCK_SEQPACKET`, which is similar to an assignment statement. Therefore, we figure out that, the anchor function `rds_sendmsg` requires a `AF_RDS` `SOCK_SEQPACKET` resource.

Matching based on Anchor Functions. Given a kernel function as the target, we match it to syscall variants based on the anchor functions associated with it from the control flow. Specifically, we perform a backward control flow analysis from the target kernel function and record all the paths. For each path, we locate all anchor functions on the path and combine their models as the model of the path. Then, we match the path to syscall variants by checking whether they share the same operation and resource model. At last, we take the merge of matched syscalls of all paths as the matching results of the target function. Note that if the backward control flow introduces an incorrect path due to the incorrect indirect call analysis, the path will not match any syscall variant because developers do not write Syzlang descriptions for a patch that does not exist. Therefore, our matching does not introduce false positives because of the inaccurate control flow analysis.

```
   /* Code snippet of File:  net/rds/af_rds.c*/
1  // Initialization of RDS module
2  static int __init rds_init(void){
3       ...
4       ret = sock_register(&rds_family_ops);
5       ...
6  }
7      ...
8  struct net_proto_family rds_family_ops = {
9       .family =        AF_RDS, //socket family
10      .create =        rds_create, //creation function
11      .owner   =       THIS_MODULE,
12 };
13       ...
14 static int rds_create(...){
15     // checking socket type
16      if (sock->type != SOCK_SEQPACKET || protocol)
17      ...
18      return __rds_create(sock, sk, protocol);}
19
20 static int __rds_create(struct socket *sock,...){
21      ... // asign function table
22      sock->ops                  = &rds_proto_ops;
23      ...
24 }
```

Figure 5: Example of Resource Modeling for Kernel Functions.

Identifying Entry Syscalls. Given a target code location, we combine the traditional control flow analysis and our matching approach to identify its entry syscalls. We first perform control flow analysis with the type-based indirect call analysis [31] to construct the control flow graph (CFG) and identify primitive syscalls that can reach the target. Then we model kernel functions with the constructed CFG and match the target function with syscall variants. At last, we take the intersection of the reachable primitive syscalls and the matched syscall variants as the entry syscalls for the target location, which prunes the infeasible analysis results.

4.2 Syscall Dependency Inference

In addition to the entry syscall that serves as the entry point, triggering the target site requires other syscalls to provide the entry syscall with resources or to set the kernel status. We refer to these system calls as related syscalls. In general, we determine related syscalls by inferring the syscalls that the entry syscall depends on via analyzing the creation and use of resources. Since the result of static inference is not sound, SyzDirect balances the use of the inferred related syscalls with the exploration of other syscalls during the fuzzing process (see §4.5).

Specifically, we adopt the static learning algorithm proposed by Healer [40] to identify syscalls that have explicit relations with the entry syscall. The algorithm first analyzes the entry syscall's input parameters and extracts the parameters' resource types. Then it recognizes the syscalls that can generate these resource types as related syscalls by analyzing the return value type of all syscalls. Since Syzlang supports inheritance between resource types, we also take the inheritance relationships when analyzing the return value type. In some cases, the resource parameter required by the target system call is of a very generic type, resulting in hundreds of related system calls being recognized by the above steps. Therefore, when too many related syscalls are identified (e.g., greater than 10), we only keep related syscalls in the same module as the entry syscall.

4.3 Syscall Argument Refinement

After determining the entry syscall, we further identify the variant-independent conditions required to reach the target location on the syscall arguments and refine the entry syscall's argument description. Due to the complexity of performing field-sensitive condition analysis of syscall parameters in the kernel, we propose a light-weight method to identify conditions on arguments instead of performing precondition analysis [9, 12, 33] or symbolic execution [11, 36].

Our light-weight condition identification consists of two steps. First, given a target location and one of its entry syscalls, we identify two kinds of conditions that dominate the target location. Similar to existing data condition sensitive fuzzing [7, 17], we focus on the conditions of comparison with constants (e.g., magic numbers) and extract the constant value as well as the literal of the constant. In addition, we also take the resource condition which affects indirect calls into consideration. That is, a syscall variant might take some sub-resource (e.g., a control message to send) as its argument, and the type of sub-resource determines the target of indirect call in a similar way to the motivation case described in §2.2. To model the resource conditions, we analyze the sub-resource registering and represent a sub-resource with the constants associated with it, as what we do for resource modeling in §4.1.

After extracting the conditions on the arguments from the kernel code, we match each extracted condition with the argument conditions included in the Syzlang description of the entry syscall. We perform condition matching by inspecting whether the literal of the extracted condition appears in the definition of any argument (as well as its recursively nested fields) and whether the value of the literal in the kernel code is the same as that in the Syzlang description. Our literal/value-based matching takes advantage of the fact that the well-written Syzlang descriptions clearly define the syscall parameters (including object parameters) as well as the constants in parameter definitions and the naming of constants refers to the linux header files. Once a condition is matched, we obtain constraints about which field of the argument should be set to which value/type for triggering the target site. That is, the other values/types for the constrained field are useless for DGF and should be pruned from the description. Thus, we refine the syscall argument description by removing the definition of these useless values/types from the description.

For example, as shown in Figure 6, the second parameter of `sendmsg$rds` is a `msghrd_rds` object and its structure is defined in line 3 to line 7. One of its nested structures, `cmsghrd_rds` is defined in line 9 to line 13. `cmsghrd_rds` is a union and could be any of the legal control message objects. Syzlang further defines seven legal control message type from line 10 to line 12 and specifies the value of the type field (RDS_CMSG_RDMA_ARGS,

```
1  sendmsg$rds(fd sock_rds, msg ptr[in, msghdr_rds], ...)
2  //definition of the msghdr_rds structure
3  msghdr_rds {
4       ...
5       ctrl    ptr[in, array[cmsghdr_rds], opt]
6       ...
7  }
8  //cmsghdr_rds is a union structure, which can be any types
defined in the following list
9  cmsghdr_rds [
10      rdma_args      cmsghdr_rds_t[RDS_CMSG_RDMA_ARGS, ...]
11      rdma_dest      cmsghdr_rds_t[RDS_CMSG_RDMA_DEST, ...]
12      ...
13 ]
```

Figure 6: Example of Argument Refinement.

RDS_CMSG_RDMA_DEST, etc.) for each message type. From the condition at line 6 of Figure 1, we extract a potential argument condition named "RDS_CMSG_RDMA_ARGS", and it matches the argument description in line 10 of Figure 6. Thus we know reaching `rds_rdma_extra_size` requires `sendmsg$rds` take a "RDS_CMSG_RDMA_ARGS" message and other types of message objects are useless. At last, we refine the argument definition by removing the definition of other message objects (i.e., line 11 - line 12) from the definition of `cmsghdr_rds`.

4.4 Distance Calculation and Instrumentation

SyzDirect mainly follows AFLGo's method to calculate the static basic block level distance and makes some adaptations for the Linux kernel. First, SyzDirect constructs the call graph (CG) of the Linux kernel with the MLTA algorithm [31] which handles the indirect calls based on type-based analysis. Based on the constructed CG, SyzDirect performs reachability analysis for the target site to find out the reachable functions and unreachable functions. For basic blocks in unreachable functions, their distance to the target site is recorded as infinity. Then, SyzDirect performs intra-procedural analysis to construct the CFG for each reachable function. Based on the CFG and CG, SyzDirect follows the formula proposed in AFLGo to calculate the basic block level distance d_b for each basic block in reachable functions. Compared to the original implementation of AFLGo, we introduce the reachability analysis to reduce the overhead caused by the unrelated code during distance calculation.

In addition, we compute several utilities that are used to facilitate the directedness in kernel fuzzing based on the block-level distance.

(1) **Syscall level distance** describes the distance of the execution path of a syscall to the target site. The syscall distance is computed by taking the shortest distance of all the basic blocks executed by the syscall.
(2) **Seed level distance** describes the distance of the input seed execution to the target code. We use the minimum syscall distance of all syscalls included in the input seed as the seed level distance.
(3) **Template level distance** describes the distance of all the seeds generated by the template to the target site. We take the average of the top 5 short seed distances generated from a template to calculate the template level distance.

We modify the KCOV module [23] to instrument the kernel to track the basic block level distance. Since the KCOV module monitors the execution of an entire syscall, we also modified the KCOV module to dynamically calculate the syscall distance. After receiving distance feedback from the KCOV, the fuzzer calculates the seed level distance and template level distance.

4.5 Directed Kernel Fuzzing with Template Guidance

SyzDirect generates templates based on the static analysis results. In particular, SyzDirect generates a template for each pair of entry syscall and its related syscall with the refined argument description. In the directed fuzzing process, SyzDirect utilizes the generated templates and runtime distance feedback to guide the fuzzer. In general, the mutation and scheduling strategies of the directed fuzzer follow two principles to improve its effectiveness.

(1) **Template guided mutation.** The template provides the necessary syscall sequence and important argument conditions for triggering the target site, that the static analysis identifies. Therefore, SyzDirect performs seed mutation under the guidance of templates. That is, it tends to generate seed inputs that not only contain the syscall sequence indicated by the template but also satisfy the parameter constraints indicated by the template.
(2) **Distance guided scheduling.** Similar to the existing directed fuzzers [10, 14], SyzDirect schedules the seed priority based on the distance feedback. Seeds that are closer to the target site have a greater probability to be mutated. In addition, since static analysis may provide multiple templates, the fuzzer also needs to prioritize the templates based on the distance information.

Based on these principles, we develop new seed selection policies and seed mutation strategies and introduce the template selection mechanism. In the following, we introduce these strategies in detail.

Initial Corpus Filtering and Generation. The fuzzer takes an initial corpus as the initial seeds. Since the initial corpus has a great impact on the performance of the fuzzer [34, 37], SyzDirect filters out the initial seeds that are of no use for reaching the target site. Specifically, SyzDirect analyzes the seeds in the initial corpus and matches them with the templates provided by the static analysis. If the syscall sequence of a seed contains the sequence provided by a certain template, SyzDirect keeps this seed. If not, SyzDirect rejects the seed.

In addition, SyzDirect performs initial seed generation if there are no valuable initial seeds. Given a template, SyzDirect generates several programs which consist of the syscalls provided by the template and leverage sSyzkaller's argument generation to fulfill their arguments. In addition, SyzDirect ensures that the resource generated by the related syscall is provided as the parameter to the entry syscall when generating the seeds.

Seed Preservation and Scheduling. As introduced in §4.4, We define the distance of a test seed as the shortest distance of all syscalls contained in that seed. As a directed fuzzer, SyzDirect regards a test case as interesting and preserves it as a seed if the test case has new edge coverage or achieves a shorter distance. SyzDirect schedules the seed in two aspects. First, in the seed selection process, SyzDirect prioritizes seeds with shorter distances because a shorter distance indicates that the seed is more beneficial for triggering the target site. Second, SyzDirect introduces power scheduling into the fuzzer which assigns different mutation chances to the test cases. A test case that is very close to the target site has more chances to trigger the target and should be assigned with more energy. Specifically, SyzDirect migrates the annealing-based power scheduling algorithm used by AFLGo [10].

Seed Mutation and Generation. SyzDirect utilizes templates to guide the seed mutation process in two dimensions: syscall mutation and parameters mutation. From the syscall perspective, for a given template, SyzDirect ensures that the test case generated by mutation contains both the entry syscall and the related syscall provided by the template. Specifically, SyzDirect checks whether the syscall sequence of the mutated test case matches a certain template. If there is no matching template, SyzDirect selects a template and inserts the syscall sequence of the selected template at the end of the test case. From the parameter perspective, SyzDirect ensures that the mutation process adheres to the parameter constraints in the template. In addition, since mutating the parameters of the target system call is more valuable for triggering the target, SyzDirect increases the parameter mutation probability for the entry syscall.

When the seed queue is empty, Syzkaller might generate a new seed directly. Similar to the adjustments made to the mutation strategy, we also modified the seed generation strategy. Specifically, SyzDirect likewise selects a template, generates a new program, and then inserts the syscalls from the template at the end of that program. During the generation process, SyzDirect ensures that the generated seed conforms to the parameter constraints of the template.

Template Selection. When the static analysis provides multiple templates, the fuzzer needs to select a template from them during the seed mutation. Since a shorter distance indicates that the template is of higher quality compared to other templates, SyzDirect also prioritizes the templates based on the template distance. Specifically, SyzDirect assigns weights values to each template based on distance and then calculates the probability of a template t being selected based on the weights.

$$Probability(t) = \frac{weight_t}{\sum_{i=1}^{N} weight_i}$$

5 EVALUATION

In this section, we comprehensively evaluate SyzDirect in various aspects. First, we evaluate the performance of SyzDirect in two significant application scenarios of directed fuzzing—bug reproduction and patch testing, which is an important criterion for measuring DGFs' capabilities. We also compare SyzDirect with Syzkaller, SyzGo (the kernel port of AFLGo) and GREBE [28]. Second, we evaluated the the contribution of each key component of SyzDirect, including the syscall identification (i.e., the combination of entry point identification and syscall dependency inference), syscall argument refinement, and distance guidance. Third, we evaluate the accuracy of the static analysis, considering its significant impact on the performance of the directed fuzzer.

In summary, we aim to answer the following questions:

- RQ1: How does SyzDirect perform in reproducing the target bugs?

- RQ2: What is the performance of SyzDirect in patch testing?
- RQ3: How does each component contribute to SyzDirect's performance respectively?
- RQ4: How effective is the static analysis of SyzDirect at extracting entry syscalls/arguments constraints?

5.1 Evaluation Setup

Evaluation Dataset. We construct two datasets to evaluate the performance of SyzDirect in the application scenario of bug reproduction and patch testing.

- **Known bugs.** We collect disclosed bugs that affect stable Linux kernel versions (i.e., v5.10-v6.2) from the Syzbot platform as the bug dataset. In order to mitigate the influence of the stability of bug triggering on our evaluation, we filter out the bugs that can not be steadily reproduced with the reproducer provided by Syzbot.
  To ensure the diversity of the bug dataset, we follow two different strategies to select bugs from the remaining bugs. ❶ The first strategy is a purely random selection. We randomly pick 50 bugs that involve different bug types including WARNING, GPF, panic, etc. ❷ The second strategy is to select security-related bugs. In particular, we randomly pick 50 bugs that associate with Kernel Address Sanitizer (KASAN) because memory corruption errors are the most common security bugs in the kernel. Following such selection strategies, the dataset covers different types of kernel bugs and includes a lot of security-related flaws. Finally, the dataset consists of 100 bugs and we evaluate SyzDirect's ability for bug reproduction on them.
- **Patches.** To ensure patch diversity, we take two different approaches to collect benign and faulty patches respectively. ❶ We randomly collected 33 patch fixes from the Syzbot platform. We manually confirmed that all 33 patches have correctly fixed the bugs and we took them as benign patches. ❷ As there is no publicly available dataset of faulty patches for the Linux kernel, we manually located faulty patches in the Linux kernel repository by reviewing the commit messages. Specifically, we inspected the "fixes" field of a commit, which points to the bug-introducing commit, and checked whether the bug-introducing commit was also a bug-fix commit. If it was, the bug-introducing commit was a faulty patch. In the end, we inspected commits from Linux kernel v5.10 to v5.18 and randomly selected 29 samples from the manually collected faulty patches, and the patch dataset consists of 62 (33+29) patches in all.

Due to the limited space of the paper, these two datasets are shown in detail in our github repository.

Evaluation Metrics. For patch testing and bug reproducing, we mainly focus on the number of cases each fuzzer could cover within the time limit. That is, the number of patches the fuzzer can successfully touch and the number of bugs it can successfully reproduce. For each case, we use the *hitting-round* and *Time-to-Exposure (TTE)* as the evaluation metrics, which are widely used by directed fuzzing papers[10, 14]. Specifically, *hitting-round* represents the times that a fuzzer triggers the target bug or reaches the target site in several repeated experiments. *TTE* is the first time that a fuzzer completes the target task (triggering target bug and so on). We calculate the arithmetic average of *TTE* in multiple repeated experiments as μTTE. In specific, if a fuzzer does not trigger the target over the fuzzing timeout (e.g., 24 hours), the TTE of this case is regarded as the timeout.

Baseline Fuzzers. We compare SyzDirect with three fuzzers: Syzkaller[18], SyzGo and GREBE [28].

- **Syzkaller** is the most widely used coverage-guided kernel fuzzer. We choose Syzkaller as a baseline for non-directed fuzzers.
- **SyzGo** is a fuzzer where we adopt the method of AFLGo[10] (one of the state-of-art DGFs) to Syzkaller and we take it as a baseline for directed fuzzers. In particular, SyzGo collects the seed distance via the method introduced in §4.4 and leverages AFLGo's annealing-based power scheduling to prioritize seeds.
- **GREBE** is the most related work of SyzDirect. It takes a directed fuzzing approach to explore more error behaviors of a kernel bug. In particular, we made two adjustments to migrate GREBE to our evaluation tasks. First, GREBE requires the PoC program as its initial corpus which is not available in bug reproduction and patch testing. Thus GREBE takes the default corpus provided by Syzkaller as its initial seeds in our evaluation. Second, the original GREBE's analysis failed in some cases of our dataset because it does not fit the latest version of KASAN. Thus, we extended GREBE's bug report parsing and fixed these failures.

We use Syzkaller, SyzGo and GREBE as baselines to show that SyzDirect is far more efficient than existing directed and non-directed fuzzers.

Environment and Configuration. In all experiments, we utilize LLVM to compile the kernel with the configuration files provided by Syzbot. Since the initial seed corpus greatly impacts the effectiveness of fuzzers[34, 37], we use the same initial seeds for the three fuzzers, provided by the default setting of Syzkaller. All experiments are conducted on a server machine with 144 Intel(R) Xeon(R) Gold 6254 CPUs (3.10GHz) and 384 GB RAM, running a 64-bit Ubuntu 18.04 LTS system.

5.2 RQ1: Bug Reproduction

In this experiment, we apply SyzDirect as well as three baseline fuzzers to reproduce bugs in the known bug dataset with their crash reports. For each bug, we analyze the stack trace of the crash reports and then manually determine the target code location for SyzDirect and SyzGo. GREBE takes the crash reports as input and automatically identifies the critical objects. For each target, we set a fuzzing time limit of 24 hours for SyzDirect and SyzGo. Since Syzkaller is not a directed fuzzer, we give it 48 hours to have a fair comparison. Each experiment is repeated 10 times to reduce statistical errors. In addition, we run GREBE for 7 days for each case, as the authors of GREBE suggested in the paper. Since it is inaccurate to confirm whether the target bug is reproduced by matching the crash title[32], we manually checked all crashes generated during fuzzing to determine if the target bug was successfully reproduced by a fuzzer.

Comparison with Syzkaller. During the experiment, Syzkaller and SyzDirect reproduced 43 bugs. We present the detailed results in Table 1 and summarize the 24-hour result in the Venn diagram of Figure 7. In particular, SyzDirect and Syzkaller$_{24h}$ reproduced 42 and 10 known bugs, respectively. SyzDirect covered all bugs

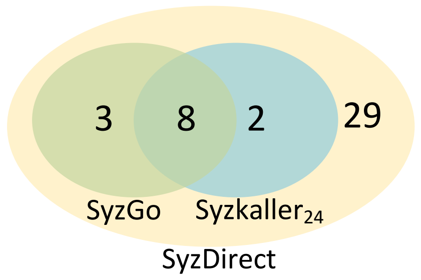

Figure 7: The Venn diagram of 24-hours bug reproducing results.

reproduced by Syzkaller$_{24h}$. Even though we ran Syzkaller for 48 hours, Syzkaller$_{48h}$ only covered two more bugs that SyzDirect could not. In addition, SyzDirect achieved a 154.3x speed up compared to Syzkaller on the 10 bugs covered by both of them. This result demonstrates that SyzDirect significantly outperforms the existing generic kernel fuzzer.

Comparison with SyzGo. The performance of SyzGo is also presented in Table 1 and Figure 7. In particular, SyzGo reproduced 11 known bugs, all of which could be reproduced by SyzDirect. In addition, on the 10 bugs, SyzDirect achieved a 81.9x speed up compared to SyzGo. Surprisingly, though SyzGo is a directed fuzzer, it does not fare much better than Syzkaller. This demonstrates that distance guiding alone is not effective enough in narrowing down the exploration space of kernel fuzzing. In contrast, SyzDirect introduces new approaches and techniques and significantly outperforms SyzGo.

Comparison with GREBE. We present the comparison between SyzDirect and GREBE in Table 2. In particular, GREBE reproduces 19 bugs, 9 of which can not be reproduced by SyzDirect in 24 hours. Interestingly, 5 of the 9 bugs are related to KASAN. We think this is because the novel object-driven fuzzing proposed by GREBE is well-suited to reproduce memory corruption bugs. Meanwhile, SyzDirect covered 32 bugs that GREBE couldn't reproduce even after fuzzing for 7 days. It shows that in more cases, SyzDirect's scheme can reduce the exploration space more effectively than GREBE. In short, SyzDirect can cover a larger number of bugs while GREBE is good at dealing with some KASAN bugs, which is hard for SyzDirect. GREBE and SyzDirect could complement each other.

Table 2: Comparison between SyzDirect and GREBE on reproducing bugs.

Reproduced by SyzDirect Only	Reproduced by both SyzDirect and GREBE	Reproduced by GREBE Only
32	10	9

5.3 RQ2: Patch Testing

In this experiment, we apply SyzDirect to test patches in the patch dataset and compare our approach with SyzGo and Syzkaller.

Table 1: The performance of SyzDirect, SyzGo and Syzkaller in bug reproducing. We present the results of running Syzkaller for 24 hours and 48 hours, respectively. For a fair comparison, we only calculate the time Speedup of SyzDirect compared to Syzkaller$_{24h}$.

Bug ID	Fuzzer	Runs	μTTE(h)	Speedup
1	Syzdirect	1/10	22.82	/
	SyzGo	0/10	24.00	1.05
	Syzkaller$_{24h}$	0/10	24.00	1.05
	Syzkaller$_{48h}$	0/10	48.00	/
2	Syzdirect	10/10	2.37	/
	SyzGo	0/10	24.00	10.14
	Syzkaller$_{24h}$	0/10	24.00	10.14
	Syzkaller$_{48h}$	0/10	48.00	/
3	Syzdirect	10/10	1.25	/
	SyzGo	0/10	24.00	19.20
	Syzkaller$_{24h}$	0/10	24.00	19.20
	Syzkaller$_{48h}$	0/10	48.00	/
4	Syzdirect	10/10	0.02	/
	SyzGo	0/10	24.00	1270.59
	Syzkaller$_{24h}$	1/10	23.15	1225.59
	Syzkaller$_{48h}$	1/10	42.80	/
5	Syzdirect	5/10	15.28	/
	SyzGo	0/10	24.00	1.57
	Syzkaller$_{24h}$	0/10	24.00	1.57
	Syzkaller$_{48h}$	0/10	48.00	/
6	Syzdirect	10/10	7.70	/
	SyzGo	3/10	22.00	2.86
	Syzkaller$_{24h}$	2/10	23.67	3.07
	Syzkaller$_{48h}$	2/10	42.87	/
9	Syzdirect	3/10	20.22	/
	SyzGo	0/10	24.00	1.19
	Syzkaller$_{24h}$	0/10	24.00	1.19
	Syzkaller$_{48h}$	0/10	48.00	/
10	Syzdirect	1/10	23.42	/
	SyzGo	0/10	24.00	1.02
	Syzkaller$_{24h}$	0/10	24.00	1.02
	Syzkaller$_{48h}$	0/10	48.00	/
11	Syzdirect	0/10	24.00	/
	SyzGo	0/10	24.00	1.00
	Syzkaller$_{24h}$	0/10	24.00	1.00
	Syzkaller$_{48h}$	1/10	47.47	/
13	Syzdirect	1/10	22.33	/
	SyzGo	0/10	24.00	1.07
	Syzkaller$_{24h}$	0/10	24.00	1.07
	Syzkaller$_{48h}$	0/10	48.00	/
14	Syzdirect	10/10	0.01	/
	SyzGo	10/10	7.55	604.00
	Syzkaller$_{24h}$	10/10	2.83	226.67
	Syzkaller$_{48h}$	10/10	2.83	/
15	Syzdirect	10/10	0.14	/
	SyzGo	1/10	23.37	162.08
	Syzkaller$_{24h}$	0/10	24.00	166.47
	Syzkaller$_{48h}$	0/10	48.00	/
17	Syzdirect	10/10	1.85	/
	SyzGo	2/10	22.95	12.41
	Syzkaller$_{24h}$	2/10	22.88	12.37
	Syzkaller$_{48h}$	2/10	42.08	/
18	Syzdirect	9/10	3.70	/
	SyzGo	9/10	9.23	2.50
	Syzkaller$_{24h}$	3/10	20.97	5.67
	Syzkaller$_{48h}$	5/10	35.35	/
19	Syzdirect	10/10	2.77	/
	SyzGo	0/10	24.00	8.67
	Syzkaller$_{24h}$	0/10	24.00	8.67
	Syzkaller$_{48h}$	0/10	48.00	/
21	Syzdirect	10/10	7.63	/
	SyzGo	1/10	22.95	3.01
	Syzkaller$_{24h}$	0/10	24.00	3.14
	Syzkaller$_{48h}$	1/10	47.70	/
23	Syzdirect	3/10	18.87	/
	SyzGo	0/10	24.00	1.27
	Syzkaller$_{24h}$	0/10	24.00	1.27
	Syzkaller$_{48h}$	0/10	48.00	/
24	Syzdirect	7/10	10.93	/
	SyzGo	0/10	24.00	2.20
	Syzkaller$_{24h}$	0/10	24.00	2.20
	Syzkaller$_{48h}$	0/10	48.00	/
25	Syzdirect	10/10	1.13	/
	SyzGo	0/10	24.00	21.18
	Syzkaller$_{24h}$	0/10	24.00	21.18
	Syzkaller$_{48h}$	1/10	46.03	/
33	Syzdirect	10/10	0.64	/
	SyzGo	1/10	23.30	36.64
	Syzkaller$_{24h}$	0/10	24.00	37.75
	Syzkaller$_{48h}$	0/10	48.00	/
34	Syzdirect	1/10	22.70	/
	SyzGo	0/10	24.00	1.06
	Syzkaller$_{24h}$	0/10	24.00	1.06
	Syzkaller$_{48h}$	0/10	48.00	/
37	Syzdirect	2/10	21.60	/
	SyzGo	0/10	24.00	1.11
	Syzkaller$_{24h}$	0/10	24.00	1.11
	Syzkaller$_{48h}$	0/10	48.00	/
43	Syzdirect	0/10	24.00	/
	SyzGo	0/10	24.00	1.00
	Syzkaller$_{24h}$	0/10	24.00	1.00
	Syzkaller$_{48h}$	1/10	47.47	/
48	Syzdirect	6/10	14.37	/
	SyzGo	5/10	19.23	1.34
	Syzkaller$_{24h}$	5/10	20.33	1.42
	Syzkaller$_{48h}$	5/10	32.33	/
49	Syzdirect	1/10	23.70	/
	SyzGo	0/10	24.00	1.01
	Syzkaller$_{24h}$	0/10	24.00	1.01
	Syzkaller$_{48h}$	0/10	48.00	/
50	Syzdirect	1/10	21.83	/
	SyzGo	0/10	24.00	1.10
	Syzkaller$_{24h}$	0/10	24.00	1.10
	Syzkaller$_{48h}$	0/10	48.00	/
55	Syzdirect	8/10	8.87	/
	SyzGo	6/10	14.17	1.60
	Syzkaller$_{24h}$	6/10	16.37	1.85
	Syzkaller$_{48h}$	7/10	24.17	/
56	Syzdirect	9/10	6.32	/
	SyzGo	0/10	24.00	3.80
	Syzkaller$_{24h}$	0/10	24.00	3.80
	Syzkaller$_{48h}$	0/10	48.00	/
59	Syzdirect	10/10	2.62	/
	SyzGo	1/10	22.25	8.50
	Syzkaller$_{24h}$	1/10	23.42	8.95
	Syzkaller$_{48h}$	2/10	44.80	/
60	Syzdirect	10/10	0.11	/
	SyzGo	10/10	7.50	66.50
	Syzkaller$_{24h}$	10/10	6.32	56.01
	Syzkaller$_{48h}$	10/10	6.32	/
62	Syzdirect	10/10	0.77	/
	SyzGo	0/10	24.00	31.05
	Syzkaller$_{24h}$	0/10	24.00	31.05
	Syzkaller$_{48h}$	0/10	48.00	/
65	Syzdirect	2/10	19.25	/
	SyzGo	0/10	24.00	1.25
	Syzkaller$_{24h}$	0/10	24.00	1.25
	Syzkaller$_{48h}$	0/10	48.00	/
68	Syzdirect	1/10	21.63	/
	SyzGo	0/10	24.00	1.11
	Syzkaller$_{24h}$	0/10	24.00	1.11
	Syzkaller$_{48h}$	0/10	48.00	/
70	Syzdirect	1/10	22.25	/
	SyzGo	0/10	24.00	1.08
	Syzkaller$_{24h}$	0/10	24.00	1.08
	Syzkaller$_{48h}$	0/10	48.00	/
71	Syzdirect	2/10	20.45	/
	SyzGo	0/10	24.00	1.17
	Syzkaller$_{24h}$	0/10	24.00	1.17
	Syzkaller$_{48h}$	0/10	48.00	/
74	Syzdirect	2/10	21.62	/
	SyzGo	0/10	24.00	1.11
	Syzkaller$_{24h}$	0/10	24.00	1.11
	Syzkaller$_{48h}$	0/10	48.00	/
75	Syzdirect	5/10	12.08	/
	SyzGo	0/10	24.00	1.99
	Syzkaller$_{24h}$	0/10	24.00	1.99
	Syzkaller$_{48h}$	0/10	48.00	/
77	Syzdirect	10/10	3.37	/
	SyzGo	0/10	24.00	7.13
	Syzkaller$_{24h}$	0/10	24.00	7.13
	Syzkaller$_{48h}$	0/10	48.00	/
80	Syzdirect	1/10	22.55	/
	SyzGo	0/10	24.00	1.06
	Syzkaller$_{24h}$	0/10	24.00	1.06
	Syzkaller$_{48h}$	0/10	48.00	/
82	Syzdirect	1/10	22.23	/
	SyzGo	0/10	24.00	1.08
	Syzkaller$_{24h}$	0/10	24.00	1.08
	Syzkaller$_{48h}$	0/10	48.00	/
86	Syzdirect	10/10	0.31	/
	SyzGo	0/10	24.00	76.87
	Syzkaller$_{24h}$	0/10	24.00	76.87
	Syzkaller$_{48h}$	0/10	48.00	/
87	Syzdirect	2/10	21.45	/
	SyzGo	0/10	24.00	1.12
	Syzkaller$_{24h}$	0/10	24.00	1.12
	Syzkaller$_{48h}$	0/10	48.00	/
95	Syzdirect	8/10	9.42	/
	SyzGo	0/10	24.00	2.55
	Syzkaller$_{24h}$	0/10	24.00	2.55
	Syzkaller$_{48h}$	0/10	48.00	/
99	Syzdirect	7/10	10.62	/
	SyzGo	0/10	24.00	2.26
	Syzkaller$_{24h}$	2/10	22.32	2.10
	Syzkaller$_{48h}$	2/10	41.52	/

Table 3: The performance of SyzDirect, SyzGo and Syzkaller in patch testing. We present the results of running Syzkaller for 24 hours and 48 hours, respectively. For a fair comparison, we only calculate the time speedup of SyzDirect compared to Syzkaller$_{24h}$.

Patch ID	Fuzzer	Runs	μTTE(h)	Speedup
2	Syzdirect	10/10	0.46	/
	SyzGo	2/10	20.37	44.54
	Syzkaller$_{24h}$	1/10	22.88	50.05
	Syzkaller$_{48h}$	2/10	44.07	/
3	Syzdirect	10/10	1.78	/
	SyzGo	0/10	24.00	13.46
	Syzkaller$_{24h}$	1/10	22.88	12.83
	Syzkaller$_{48h}$	2/10	44.07	/
4	Syzdirect	0/10	24.00	/
	SyzGo	0/10	24.00	1.00
	Syzkaller$_{24h}$	0/10	24.00	1.00
	Syzkaller$_{48h}$	1/10	46.80	/
6	Syzdirect	10/10	1.60	/
	SyzGo	0/10	24.00	15.00
	Syzkaller$_{24h}$	1/10	22.40	14.00
	Syzkaller$_{48h}$	3/10	42.40	/
7	Syzdirect	10/10	0.0058	/
	SyzGo	8/10	11.03	1891.43
	Syzkaller$_{24h}$	9/10	10.03	1720.00
	Syzkaller$_{48h}$	9/10	12.43	/
8	Syzdirect	10/10	0.0067	/
	SyzGo	10/10	10.38	1557.50
	Syzkaller$_{24h}$	9/10	11.63	1745.00
	Syzkaller$_{48h}$	9/10	14.03	/
9	Syzdirect	10/10	0.0067	/
	SyzGo	10/10	4.78	717.50
	Syzkaller$_{24h}$	9/10	7.43	1115.00
	Syzkaller$_{48h}$	9/10	9.83	/
10	Syzdirect	9/10	9.80	/
	SyzGo	7/10	13.12	1.34
	Syzkaller$_{24h}$	8/10	12.80	1.31
	Syzkaller$_{48h}$	9/10	17.02	/
11	Syzdirect	10/10	0.07	/
	SyzGo	5/10	17.05	256.82
	Syzkaller$_{24h}$	5/10	16.73	252.05
	Syzkaller$_{48h}$	8/10	22.85	/
12	Syzdirect	10/10	0.02	/
	SyzGo	10/10	0.0028	0.12
	Syzkaller$_{24h}$	0/10	24.00	1016.47
	Syzkaller$_{48h}$	0/10	48.00	/
13	Syzdirect	10/10	0.0119	/
	SyzGo	9/10	10.05	841.40
	Syzkaller$_{24h}$	9/10	11.23	940.47
	Syzkaller$_{48h}$	9/10	13.63	/
14	Syzdirect	10/10	0.0056	/
	SyzGo	10/10	1.17	210.00
	Syzkaller$_{24h}$	9/10	3.97	714.00
	Syzkaller$_{48h}$	9/10	6.37	/
15	Syzdirect	10/10	0.0058	/
	SyzGo	10/10	0.0131	2.24
	Syzkaller$_{24h}$	9/10	2.40	411.43
	Syzkaller$_{48h}$	9/10	4.80	/
16	Syzdirect	6/10	9.65	/
	SyzGo	5/10	14.32	1.48
	Syzkaller$_{24h}$	4/10	16.80	1.74
	Syzkaller$_{48h}$	7/10	27.30	/
18	Syzdirect	2/10	20.10	/
	SyzGo	0/10	24.00	1.19
	Syzkaller$_{24h}$	0/10	24.00	1.19
	Syzkaller$_{48h}$	0/10	48.00	/
20	Syzdirect	9/10	8.63	/
	SyzGo	2/10	22.92	2.65
	Syzkaller$_{24h}$	2/10	21.88	2.53
	Syzkaller$_{48h}$	5/10	37.12	/
21	Syzdirect	2/10	22.72	/
	SyzGo	0/10	24.00	1.06
	Syzkaller$_{24h}$	0/10	24.00	1.06
	Syzkaller$_{48h}$	0/10	48.00	/
22	Syzdirect	10/10	0.0053	/
	SyzGo	10/10	0.03	5.32
	Syzkaller$_{24h}$	9/10	2.42	457.89
	Syzkaller$_{48h}$	9/10	4.82	/
23	Syzdirect	2/10	22.52	/
	SyzGo	0/10	24.00	1.07
	Syzkaller$_{24h}$	1/10	23.25	1.03
	Syzkaller$_{48h}$	2/10	44.77	/
24	Syzdirect	10/10	1.63	/
	SyzGo	0/10	24.00	14.69
	Syzkaller$_{24h}$	1/10	22.18	13.58
	Syzkaller$_{48h}$	1/10	43.78	/
25	Syzdirect	10/10	0.0044	/
	SyzGo	10/10	0.03	5.81
	Syzkaller$_{24h}$	9/10	2.42	543.75
	Syzkaller$_{48h}$	9/10	4.82	/
26	Syzdirect	10/10	0.02	/
	SyzGo	2/10	22.88	1373.00
	Syzkaller$_{24h}$	3/10	21.57	1294.00
	Syzkaller$_{48h}$	3/10	38.37	/
27	Syzdirect	10/10	0.0039	/
	SyzGo	2/10	20.37	5237.14
	Syzkaller$_{24h}$	3/10	21.57	5545.71
	Syzkaller$_{48h}$	5/10	37.28	/
28	Syzdirect	10/10	1.75	/
	SyzGo	0/10	24.00	13.71
	Syzkaller$_{24h}$	0/10	24.00	13.71
	Syzkaller$_{48h}$	0/10	48.00	/
29	Syzdirect	1/10	23.78	/
	SyzGo	0/10	24.00	1.01
	Syzkaller$_{24h}$	0/10	24.00	1.01
	Syzkaller$_{48h}$	0/10	48.00	/
31	Syzdirect	10/10	0.0133	/
	SyzGo	7/10	14.93	1120.00
	Syzkaller$_{24h}$	6/10	13.58	1018.75
	Syzkaller$_{48h}$	7/10	22.50	/
32	Syzdirect	10/10	0.0039	/
	SyzGo	10/10	1.18	304.29
	Syzkaller$_{24h}$	9/10	4.17	1071.43
	Syzkaller$_{48h}$	9/10	6.57	/
34	Syzdirect	1/10	23.92	/
	SyzGo	0/10	24.00	1.00
	Syzkaller$_{24h}$	0/10	24.00	1.00
	Syzkaller$_{48h}$	0/10	48.00	/
35	Syzdirect	10/10	0.0064	/
	SyzGo	10/10	2.10	328.70
	Syzkaller$_{24h}$	10/10	3.15	493.04
	Syzkaller$_{48h}$	10/10	3.15	/
36	Syzdirect	10/10	6.43	/
	SyzGo	10/10	7.37	1.15
	Syzkaller$_{24h}$	10/10	9.87	1.53
	Syzkaller$_{48h}$	10/10	9.87	/
38	Syzdirect	7/10	11.20	/
	SyzGo	8/10	12.33	1.10
	Syzkaller$_{24h}$	4/10	18.32	1.64
	Syzkaller$_{48h}$	4/10	32.72	/
39	Syzdirect	10/10	0.0042	/
	SyzGo	10/10	0.76	182.40
	Syzkaller$_{24h}$	10/10	1.88	452.00
	Syzkaller$_{48h}$	10/10	1.88	/
40	Syzdirect	10/10	0.04	/
	SyzGo	10/10	0.03	0.68
	Syzkaller$_{24h}$	10/10	0.02	0.59
	Syzkaller$_{48h}$	10/10	0.02	/
41	Syzdirect	8/10	11.82	/
	SyzGo	8/10	10.47	0.89
	Syzkaller$_{24h}$	10/10	7.07	0.60
	Syzkaller$_{48h}$	10/10	7.07	/
42	Syzdirect	0/10	24.00	/
	SyzGo	0/10	24.00	1.00
	Syzkaller$_{24h}$	0/10	24.00	1.00
	Syzkaller$_{48h}$	3/10	40.82	/
43	Syzdirect	10/10	0.0047	/
	SyzGo	10/10	0.83	176.76
	Syzkaller$_{24h}$	10/10	0.91	193.71
	Syzkaller$_{48h}$	10/10	0.91	/
44	Syzdirect	1/10	22.08	/
	SyzGo	0/10	24.00	1.09
	Syzkaller$_{24h}$	0/10	24.00	1.09
	Syzkaller$_{48h}$	0/10	48.00	/
45	Syzdirect	10/10	0.0142	/
	SyzGo	10/10	1.03	72.94
	Syzkaller$_{24h}$	10/10	0.81	56.92
	Syzkaller$_{48h}$	10/10	0.81	/
46	Syzdirect	2/10	20.23	/
	SyzGo	1/10	22.45	1.11
	Syzkaller$_{24h}$	1/10	23.85	1.18
	Syzkaller$_{48h}$	3/10	42.27	/
47	Syzdirect	10/10	1.42	/
	SyzGo	1/10	23.78	16.79
	Syzkaller$_{24h}$	1/10	23.33	16.47
	Syzkaller$_{48h}$	1/10	44.93	/
48	Syzdirect	10/10	0.0044	/
	SyzGo	10/10	0.03	6.31
	Syzkaller$_{24h}$	10/10	0.03	6.25
	Syzkaller$_{48h}$	10/10	0.03	/
50	Syzdirect	10/10	1.87	/
	SyzGo	1/10	23.18	12.42
	Syzkaller$_{24h}$	4/10	17.32	9.28
	Syzkaller$_{48h}$	6/10	28.77	/
52	Syzdirect	10/10	0.0058	/
	SyzGo	9/10	9.72	1665.71
	Syzkaller$_{24h}$	10/10	8.73	1497.14
	Syzkaller$_{48h}$	10/10	8.73	/
53	Syzdirect	10/10	0.43	/
	SyzGo	10/10	2.87	6.66
	Syzkaller$_{24h}$	10/10	3.25	7.55
	Syzkaller$_{48h}$	10/10	3.25	/
54	Syzdirect	8/10	9.93	/
	SyzGo	0/10	24.00	2.42
	Syzkaller$_{24h}$	0/10	24.00	2.42
	Syzkaller$_{48h}$	0/10	48.00	/
55	Syzdirect	1/10	23.00	/
	SyzGo	0/10	24.00	1.04
	Syzkaller$_{24h}$	0/10	24.00	1.04
	Syzkaller$_{48h}$	0/10	48.00	/
58	Syzdirect	10/10	0.0036	/
	SyzGo	7/10	15.48	4287.69
	Syzkaller$_{24h}$	8/10	15.40	4264.62
	Syzkaller$_{48h}$	10/10	16.53	/
59	Syzdirect	10/10	3.88	/
	SyzGo	2/10	23.17	5.97
	Syzkaller$_{24h}$	4/10	19.60	5.05
	Syzkaller$_{48h}$	7/10	29.35	/
60	Syzdirect	8/10	6.98	/
	SyzGo	0/10	24.00	3.44
	Syzkaller$_{24h}$	0/10	24.00	3.44
	Syzkaller$_{48h}$	0/10	48.00	/
61	Syzdirect	10/10	0.0044	/
	SyzGo	10/10	8.83	1987.50
	Syzkaller$_{24h}$	10/10	6.97	1567.50
	Syzkaller$_{48h}$	10/10	6.97	/
62	Syzdirect	10/10	0.0025	/
	SyzGo	10/10	0.0028	1.11
	Syzkaller$_{24h}$	10/10	2.63	1053.33
	Syzkaller$_{48h}$	10/10	2.63	/

The goal of patch testing is to test the patch code under different contexts to discover potential patch-related bugs. Thus we take patch coverage as the primary metric, i.e., whether the fuzzer reaches the patched code. We set the code modified by the patch commits as the target for DGF. For the patch commits which modified several positions, we manually locate the code that is the most relevant to the bug cause as the target position.

For the 62 patches, 49 can be reached by at least one fuzzer in 24 hours. As shown in Figure 8, SyzDirect covers all patches touched by other fuzzers. Specifically, SyzDirect, SyzGo, and Syzkaller$_{24h}$ cover 49, 36 and 39 patches, respectively. Compared to these fuzzers, Syzkaller$_{48h}$ can only reach one more patch. Table 3 presents the exact time when the fuzzer reaches each patch. On most patches, SyzDirect reaches the patch location faster than all the baselines, achieving an average of 680.9 speedup and 620.2 speedup compared to Syzkaller$_{24h}$ and SyzGo.

In addition, during the patch testing, SyzDirect, SyzGo, and Syzkaller discovered 15, 11, and 12 bugs, which are introduced by the known faulty patches. In particular, SyzDirect discovered 4 more faulty-patch-introduced bugs that the other two baselines could not detect. This result suggests that the enhanced directed fuzzing capabilities brought by SyzDirect might help developers find more patch-introduced bugs.

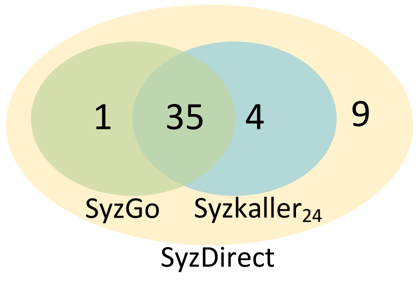

Figure 8: The Venn diagram of 24-hours patch testing results.

5.4 Failure Analysis of RQ1&RQ2

Among all the bugs and patches that SyzDirect fails to reproduce or reach in §5.2 and §5.3, we manually analyzed their PoCs from Syzbot, the static analysis results, and the fuzz logs to investigate why SyzDirect failed. In total, we summarized three reasons for these failures.

- **Incomplete dependent syscall inference (R1).** SyzDirect infers the syscalls that have an explicit dependency with the entry syscall as related syscalls, which misses the syscalls with an implicit dependency on the entry syscall. Due to the incomplete dependent syscall inference, SyzDirect failed to generate the required syscall sequence to reach the target site.
- **Difficulties in generating proper arguments (R2).** In some cases, SyzDirect generates the proper syscall sequence but fails to generate arguments that could trigger the bug or target code. Though SyzDirect leverages the extracted argument conditions to guide the argument mutation, there are still two challenges to generating proper parameters. First, SyzDirect's static analysis mainly focuses on the conditions of reaching the code instead of triggering the bug. Thus SyzDirect is unable to generate arguments that satisfy the strict constraints of triggering the bug though the bug is reached. Second, generating object arguments such as file system images remains very challenging for Syzkaller. SyzDirect inherits the limitations of Syzkaller in this aspect.
- **Lack of deep analysis for related syscall (R3).** Triggering the target bug/site requires not only the correct arguments for the entry syscall but also specific context and arguments for the related syscall. SyzDirect's static analysis does not perform deep analysis for related syscall such as argument conditions for related syscall to reach the target. As a result, SyzDirect may fail when the related syscall requires specific parameters or complicated contexts to reach the target.

As the detailed case breakdown presented in Table 4, more than half of the failures are caused by the lack of deep analysis of related syscalls. In this paper, our main technical contributions are the identification of the entry point and the argument refinement of the entry syscalls. We leave the deep analysis of related syscalls as future work. In §6, we will discuss how to improve SyzDirect.

Table 4: The cause breakdown of failure cases. R1 is incomplete dependent syscall inference; R2 is difficulties in generating proper parameters; and R3 is lack of deep analysis for related syscall.

Reasons	Bug Reproduction	Patch Testing
R1	19	2
R2	19	6
R3	20	5
Total	59	13

5.5 RQ3: Contribution of Different Components

As introduced in §4, SyzDirect's guidance on directed fuzzing comes from three components: distance feedback, syscall identification, and syscall argument refinement. To evaluate the contribution of each key component, we designed and implemented two variants of SyzDirect. ❶ We designed SyzDirect$_{NoArg}$, a variant version of SyzDirect that disables the syscall argument refinement component. In addition, the syscall parameter mutation of SyzDirect$_{NoArg}$ preserves the default implementation of Syzkaller. ❷ We designed SyzDirect$_{NoDistance}$, a variant version of SyzDirect that disables the runtime distance feedback. That is, SyzDirect$_{NoDistance}$ schedules the seed queues following the Syzkaller's default first-in-first-out order. We then reran the bug reproduction experiments using SyzDirect$_{NoArg}$ and SyzDirect$_{NoDistance}$ with a 24-hour limit and compared them with SyzDirect. In addition, we repeated the experiments 10 times for each case. Note that since syscall argument refinement is based on the syscall identification, we could not only turn off the syscall identification component. Fortunately, we can evaluate the effect of syscall identification by comparing SyzDirect$_{NoArg}$ and Syzgo. The main difference between them lies in that SyzDirect$_{NoArg}$ leverages the syscall identification.

The experimental results are shown in Figure 9. SyzDirect, SyzDirect$_{NoArg}$ and SyzDirect$_{NoDistance}$ successfully reproduced 42, 22 and 23 bugs in 24 hours, respectively. On the one hand, the full version of SyzDirect covers nearly 100% more bugs than SyzDirect$_{NoArg}$ and SyzDirect$_{NoDistance}$, proving that the syscall argument refinement and distance guidance are very effective. On the other hand, SyzDirect$_{NoArg}$ covers nearly 100% more bugs than SyzGo. The result demonstrates that distance guidance alone has little effect on DGF but can be significantly improved by utilizing the entry syscalls and dependent syscalls identified via the static analysis techniques proposed in the paper. In a word, all three key design components contribute a lot to SyzDirect.

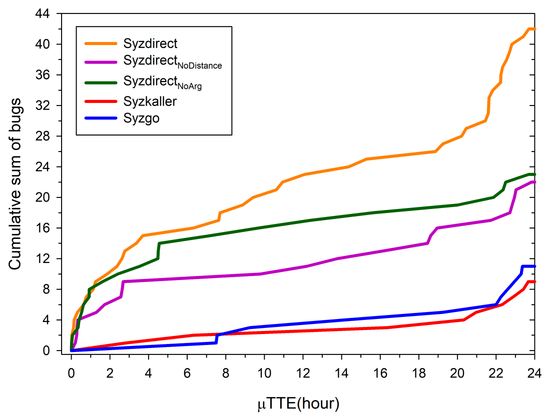

Figure 9: Comparison of bug-reproduction for SyzDirect, SyzDirect$_{NoArg}$, and SyzDirect$_{NoDistance}$. SyzGo only utilizes distance feedback. SyzDirect$_{NoArg}$ disables the syscall argument refinement of the static analysis. SyzDirect$_{NoDistance}$ disables the runtime distance feedback. SyzDirect$_{full}$ is the full version.

5.6 RQ4: Static Analysis Effectiveness

To provide a better understanding of the performance of SyzDirect's static analysis, we evaluate the the effectiveness of entry point identification, dependent syscall inference, and argument refinement in the bug reproduction dataset.

Effectiveness of entry point identification. The results of entry point identification in the 100 bugs are presented in Figure 10. The results demonstrate that SyzDirect identified an average of 3.61 entry syscalls from more than 4,000 syscalls for each bug. In particular, SyzDirect identified no more than 3 entry syscalls for 61% bugs.

It is difficult to prove that a syscall is impossible to reach the target site. We have therefore adopted an estimation method to measure the effectiveness of entry syscall identification. In particular, we extracted the entry syscall used by the PoC as the correct entry syscall and compare it with the syscalls identified by SyzDirect, which shows the lower bound of our approach. The result turns out that SyzDirect correctly located the entry syscall used by the PoCs for all 100 bugs and identified an average of 2.61 more other syscalls.. These results demonstrate that SyzDirect's static analysis is able to locate the correct entry syscall and greatly reduce the syscall entry exploration space for DGF.

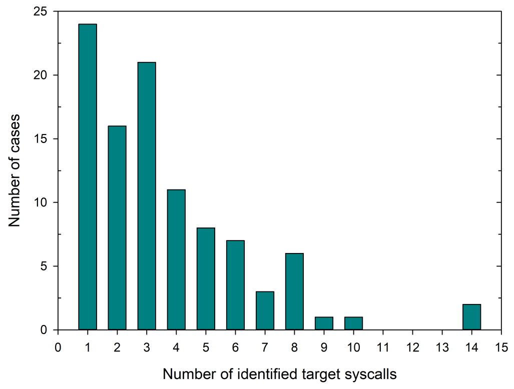

Figure 10: Distribution of the numbers of identified entry syscalls on each target bug.

Effectiveness of dependent syscall inference. SyzDirect infers the dependent syscalls for each identified entry syscall and takes the syscalls which the entry syscall depends on as related syscalls. we count the number of pairs of entry syscall and related syscall for each bug. As presented in Figure 11, 60% of the bugs have no more than 10 pairs of entry syscall and related syscall. There are six bugs that have more than 40 pairs. This is because their entry syscalls require a generic resource object (such as a TTY object) as the argument, causing Syzdirect to locate a large number of producers as related syscalls. We also compared the syscalls from the PoC with the SyzDirect's results. It turns out that SyzDirect correctly located the related syscall for 87 bugs. For the remaining 13 cases, SyzDirect fails to infer the correct related syscall because SyzDirect only considers the explicit dependencies between syscalls. More interestingly, among the 13 failed cases, SyzDirect still reproduces 6 bugs because the fuzzer generated the correct related syscall via the seed mutation.

Effectiveness of argument refinement. To evaluate the syscall argument refinement, We investigate whether SyzDirect extracts valid argument conditions for the correct entry syscall used in the PoC. In all, SyzDirect extracts argument conditions for the correct entry syscall in 59 of the 100 target bugs. After manually analyzing the bugs' triggering process, we found that the extracted conditions are necessary to reach the bug site in 54 bugs. In the other 5 bugs, the static analysis locates conditions that are not related to triggering the bug because the extracted condition is on other kernel

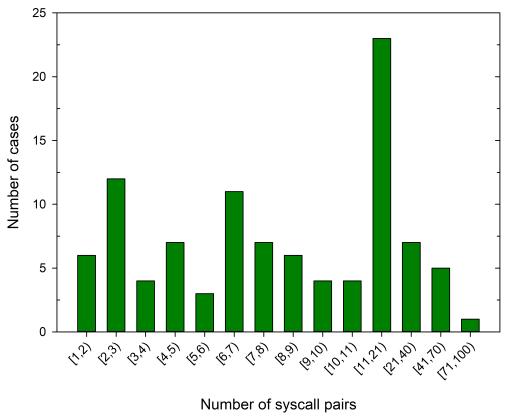

Figure 11: Distribution of the numbers of extracted pairs of entry syscall and related syscall on each target bug.

variables instead of the syscall arguments. For the 41 (100-59) bugs, SyzDirect extracts no argument conditions due to two reasons. First, some argument-related checking is very complicated and the SyzDirect's static analysis does not model such checks, thus failing to collect the conditions. Second, SyzDirect's argument refinement heavily depends on the Syzlang descriptions. For several syscalls, Syzlang description does not provide detailed argument definitions, thus the extracted conditions could not match any argument and are dropped by our analysis.

6 LIMITATIONS AND FUTURE WORK

6.1 Inaccuracy of Static Analysis

SyzDirect performs static analysis to identify various information to reach the target site. Though SyzDirect exploits the design of kernel and Syzlang descriptions to mitigate false positives of entry point identification and false negatives of argument refinement, the static analysis still has false positives and false negatives. Since we have given dynamic fuzzing flexibility to explore syscalls other than static analysis results, fuzzing can help correct part of the errors as discussed in §5.6. Completely eliminating the inaccuracies of static analysis is very challenging. On the one hand, it requires more effective building block tools to perform perfect control and data flow analysis for the kernel. On the other hand, it requires a full understanding of the kernel's semantics to match kernel functions to Syzlang descriptions.

6.2 Further Analysis of Dependent Syscalls

Dependent syscalls and their arguments also affect whether the target site can be triggered indirectly. As the first DGF for Linux kernel, we focus on addressing the main challenges of the existing works, i.e., entry point identification and argument refinement for entry syscalls. We plan to explore how to identify dependent syscalls more accurately and how to perform argument refinement for dependent syscalls in the future.

6.3 Reliance on Syzlang

As introduced in §3 and §4, the static analysis of SyzDirect does rely on manually-defined Syzlang. Fortunately, some techniques (e.g., SyzDescribe[20]) have been proposed to automatically generate syscall descriptions. With these techniques, SyzDirect could still work when Syzlang goes out of sync.

7 RELATED WORK

7.1 Directed Greybox Fuzzing

There has been lots of work on directed greybox fuzzing. We summarize the general DGF solutions in §2.1 and present their limitations in dealing with OS kernels in §2.2. We propose SyzDirect as a general DGF solution for the Linux kernel, which can serve a variety of tasks, including crash reproduction, patch testing, etc.

In addition, there is some work [28, 44] that presents some customized DGF techniques for specific tasks. KLAUS [44] aims to assess the correctness of kernel patches via directed fuzzing. In order to trigger the vulnerabilities related to incorrect patches, KLAUS's directed fuzzing mechanism not only focuses on reaching specific program sites but also considers the order and type of patch-altered read and write operations. GREBE [28] proposes a fancy method to explore more error behaviors of a kernel bug. GREBE presents a static analysis framework to identify critical kernel objects for a bug and then introduces a directed fuzzing mechanism that takes the hits of critical objects as feedback. Compared to these works, SyzDirect is a general DGF and takes the code reachability as the primary goal. Thus SyzDirect proposes several new techniques to identify syscalls and argument constraints required to reach specific program sites.

7.2 Kernel Fuzzing

Linux kernels are an important target in fuzzing. Unlike user-space applications which take the main function as the entry point, the Linux kernel has several different entry points, including system calls, I/O control handler functions, and interrupt request handlers functions. Since the syscall interface is the most widely used, many efforts are devoted to testing the kernel through the syscall interface [4, 18, 24, 34, 38, 40, 42]. Besides taking the code coverage as the feedback, many works utilize different techniques such as static analysis [34], symbolic execution [24], dynamic analysis [40] and reinforcement learning [42] to improve testing efficiency. In addition to testing the system call interface, researchers propose some novel techniques to test other channels such as direct memory access (DMA) [39], USB interface [35], and driver interruptions [21]. These works aim to cover more kernel code and bugs, while SyzDirect focuses on testing a specific code location.

7.3 Syscall Identification for Fuzzing

In addition to generic kernel fuzzing which explores all syscalls, there are also efforts that explore how to identify critical syscalls based on different fuzzing scenarios. SemFuzz [45] aims to generate proof-of-concept based on well-written bug reports. It applies natural-language processing to extract the required syscalls from bug reports and Linux git logs to guide fuzzing. GREBE [28] proposes an automated approach to identify valuable syscalls for exploring more error behaviors from the seed corpus during fuzzing. However, it requires the PoC program as its initial corpus to determine the initial range of syscalls. In a word, existing work relies on auxiliary materials such as well-written bug reports and PoC programs to facilitate the syscall identification. In contrast, as a general DGF tool, SyzDirect locates syscall the valuable syscalls for a given target site in the absence of such material.

8 CONCLUSION

In this paper, we present SyzDirect, a DGF solution for the Linux kernel. To address the unique challenges posed by the nature of Linux kernel, SyzDirect employs novel static analysis to identify the entry syscalls as well as the conditions on its arguments, which greatly narrows down the exploration space for DGF. Combining the distance-based feedback and statically extracted information, SyzDirect adjusts its seed scheduling and seed mutation to guide the fuzzer stress-test the target site. Our extensive evaluation on upstream Linux kernels shows that SyzDirect is more effective and efficient than generic kernel fuzzers and existing DGF techniques in crash reproduction and patch testing.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Key Research and Development Program (2021YFB3101200), National Natural Science Foundation of China (62172105, 62172104, 62102091, 62102093), and the Funding of Ministry of Industry and Information Technology of the People's Republic of China under Grant TC220H079. Yuan Zhang was supported in part by the Shanghai Rising-Star Program 21QA1400700 and the Shanghai Pilot Program for Basic Research-Fudan University 21TQ1400100 (21TQ012). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems and Engineering Research Center of Cyber Security Auditing and Monitoring.

REFERENCES

[1] 2017. Patch of Dirty COW Vulnerability Incomplete, Researchers Claim. https://www.securityweek.com/patch-dirty-cow-vulnerability-incomplete-researchers-claim/.
[2] 2022. K01311152: Linux kernel vulnerabilities CVE-2020-36322 and CVE-2021-28950. https://my.f5.com/manage/s/article/K01311152.
[3] 2022. net/rds: fix warn in rds_message_alloc_sgs. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ea010070d0a7497253d5a6f919f6dd107450b31a.
[4] 2022. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity..
[5] 2023. Kernel.org Bugzilla. https://bugzilla.kernel.org.
[6] 2023. Syzkaller System Call Description. https://github.com/google/syzkaller/tree/master/sys/linux.
[7] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.
[8] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. 2021. Static Detection of Unsafe DMA Accesses in Device Drivers. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 1629–1645.
[9] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: Precise Refutations for Heap Reachability. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/2491956.2462186
[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020
[11] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.

In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[12] Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: A Powerful Approach to Weakest Preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 363–374. https://doi.org/10.1145/1542476.1542517

[13] Bo Chen, Zhenkun Yang, Li Lei, Kai Cong, and Fei Xie. 2020. Automated Bug Detection and Replay for COTS Linux Kernel Modules with Concolic Execution. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 172–183. https://doi.org/10.1109/SANER48275.2020.9054797

[14] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-Box Fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2095–2108. https://doi.org/10.1145/3243734.3243849

[15] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. WindRanger: a directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering*. 2440–2451.

[16] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. 2021. Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society.

[17] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 145, 18 pages.

[18] Google. 2022. Syzkaller. https://github.com/google/syzkaller.

[19] Google. 2022. syzlang. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[20] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. 2023. SyzDescribe: Principled, Automated, Static Generation of Syscall Descriptions for Kernel Drivers. In *44rd IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 22-25, 2023*. IEEE.

[21] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. 2021. VIA: Analyzing Device Interfaces of Protected Virtual Machines. In *Annual Computer Security Applications Conference* (Virtual Event, USA) *(ACSAC '21)*. Association for Computing Machinery, New York, NY, USA, 273–284. https://doi.org/10.1145/3485832.3488011

[22] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*. 36–50. https://doi.org/10.1109/SP46214.2022.9833751

[23] kernel.org. 2022. kcov: code coverage for fuzzing. https://www.kernel.org/doc/html/v5.9/dev-tools/kcov.html.

[24] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/hfl-hybrid-fuzzing-on-the-linux-kernel/

[25] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis &amp; Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[26] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing.. In *USENIX Security Symposium*. 3559–3576.

[27] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2022. Path-sensitive and alias-aware typestate analysis for detecting OS bugs. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 859–872.

[28] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. 2022. GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs. In *2022 IEEE Symposium on Security and Privacy (SP)*. 2078–2095. https://doi.org/10.1109/SP46214.2022.9833683

[29] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. 2021. Detecting Missed Security Operations Through Differential Checking of Object-Based Similar Paths. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 1627–1644. https://doi.org/10.1145/3460120.3485373

[30] Jian Liu, Lin Yi, Weiteng Chen, Chengyu Song, Zhiyun Qian, and Qiuping Yi. 2022. LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 125–142.

[31] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1867–1881. https://doi.org/10.1145/3319535.3354244

[32] Dongliang Mu, Yuhang Wu, Yueqi Chen, Zhenpeng Lin, Chensheng Yu, Xinyu Xing, and Gang Wang. 2022. An In-depth Analysis of Duplicated Linux Kernel Bug Reports. In *Network and Distributed Systems Security (NDSS) Symposium 2022*.

[33] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-Driven Precondition Inference with Learned Features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 42–56. https://doi.org/10.1145/2908080.2908099

[34] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. Moonshine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA). USENIX Association, USA, 729–743.

[35] Hui Peng and Mathias Payer. 2020. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 144, 17 pages.

[36] David A. Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 49–64. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

[37] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA). USENIX Association, USA, 861–875.

[38] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. KAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, USA, 167–182.

[39] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *Network and Distributed System Security Symposium (NDSS)*.

[40] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3477132.3483547

[41] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. 2021. Detecting kernel refcount bugs with two-dimensional consistency checking. In *the 30th USENIX Security Symposium (Security'21)*.

[42] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael B. Abu-Ghazaleh. 2021. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2741–2758. https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng

[43] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. 2020. Precisely Characterizing Security Impact in a Flood of Patches via Symbolic Rule Comparison. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society.

[44] Yuhang Wu, Zhenpeng Lin, Yueqi Chen, Dang K Le, Dongliang Mu, and Xinyu Xing. 2023. Mitigating Security Risks in Linux with KLAUS: A Method for Evaluating Patch Correctness. In *32st USENIX Security Symposium, USENIX Security 2023*. USENIX Association.

[45] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-Based Automatic Generation of Proof-of-Concept Exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2139–2154. https://doi.org/10.1145/3133956.3134085

[46] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guoren Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V. Krishnamurthy, Trent Jaeger, and Paul L. Yu. 2022. Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society.

[47] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2255–2269. https://www.usenix.org/conference/usenixsecurity20/presentation/zong