KSG: Augmenting Kernel Fuzzing with System Call Specification Generation

Hao Sun
Tsinghua University

Yuheng Shen
Tsinghua University

Jianzhong Liu
Tsinghua University

Yiru Xu
Tsinghua University

Yu Jiang [*]
Tsinghua University

Abstract

Kernel fuzzing is a dynamic testing technique that has successfully found numerous kernel vulnerabilities. However, existing kernel fuzzers, such as Syzkaller, depend on system call specifications to generate test cases. Writing such specifications requires an immense amount of domain knowledge while being extremely laborious. Meanwhile, automated generation of the specification is still an open problem due to the complexity of the kernel, including entry function extraction and input type identification. As a result, the current amount of system call information is insufficient to test the entire kernel code base thoroughly. Syzkaller covers an average of 38% of Linux kernel code with current Syzlang specifications for a prolonged time of fuzzing.

In this paper, we propose KSG to generate system call specifications for kernel fuzzers automatically. First, it utilizes probe-based tracing to extract entry functions accurately. Then, it uses path-sensitive analysis to collect precise input types and range constraints in each execution path of entry functions. Based on the aforementioned information, KSG generates specifications in the domain language Syzlang, which is used by most kernel fuzzers. We evaluated KSG on several versions of the Linux kernel. It automatically generated 2433 unique specifications. Leveraging the newly generated specifications, Syzkaller and Moonshine achieved coverage improvements of 22% and 23% respectively. Furthermore, our approach assisted fuzzers to discover 26 previously unknown bugs, where 13 and 6 bugs were fixed and assigned with CVEs, respectively.

1 Introduction

The operating system kernel is one of the most complex components and forms the foundation of the software system. It is responsible for core functionalities, such as communication and IO of userspace applications. The security of the kernel is crucial as kernel bugs can lead to huge impacts easily, e.g., causing userspace applications to be unresponsive [24] and allowing an attacker to completely compromise a target system [22]. Fuzz testing [19] is a popular technique for automatically discovering security vulnerabilities and has already been applied to the kernel domain. For instance, Syzkaller [33], one of the most widely used kernel fuzzers, has been integrated into Linux testing pipeline. It has reported thousands of kernel bugs [31] up until now, demonstrating the effectiveness of applying fuzzing to kernel testing.

The driving force of the kernel fuzzer's bug discovery capability is system call specifications, which provide a rich set of system call information. As shown in Figure 2, a system call can accept parameters with different types based on underlying submodules' requirements. The prototype of system calls are written in C, a weakly typed language, that does not provide much information of system calls' arguments, e.g., many parameters are defined as *void**. Therefore, it is difficult for fuzzers to generate inputs that satisfy the submodules' structural constraints without additional information, resulting in low fuzzing efficiency. To address this issue, existing kernel fuzzers, such as Syzkaller, use a domain language called Syzlang [34] to encode system call specifications. Syzlang is a strongly typed language and can specialize system calls to specific submodules with precise input types, as shown in Figure 1. With this domain knowledge, fuzzing efficiency can be improved significantly. Meanwhile, the amount of specifications has a significant impact on the performance of fuzzers. Fuzzers can generate inputs to test kernel submodules that are well-encoded in the specifications, but they have difficulty reaching kernel code that is not encoded. Therefore, many researchers have to manually write numerous specifications to test the kernel thoroughly.

However, encoding system call specifications requires an immense amount of domain knowledge, resulting in significant time costs and insufficient number of specifications. Many system calls in the Linux kernel are an abstraction over corresponding functionalities of kernel submodules, which are responsible for dispatching the user input to submodules' operations. The actual types of the system call's parame-

[*]Yu Jiang is the corresponding author of this paper.

ters depend on the specific invoked submodule. Therefore, most specifications are encoded for specific submodules, e.g., *dev_loop.txt* in Syzkaller is specifications for *loop* device. In order to encode specifications manually, the following different aspects of domain knowledge are required: (1) the precise input types of system call used by the submodule; (2) the range constraints on specific input parameters; (3) the domain language used by fuzzers for specifications. However, the large amount of kernel submodules and the complexity of the input types make it difficult to encode specifications for a wide range of kernel functionalities. Consequently, most specifications are written by kernel experts, but the extensive manual effort required has caused specification shortage. Based on the coverage data reported by Syzbot's dashboard [32], Syzkaller covers an average of 38% of Linux kernel code with the current Syzlang specifications for a prolonged time of fuzzing. Several recent works perform automatic specification generation [4, 6, 11], but they are either designed for particular system calls or close-sourced scenarios. In order to further reduce manual efforts, we need to generate specifications for more system calls and their corresponding submodules.

Source code analysis can be used to generate specifications effectively, but several challenges need to be addressed due to the complexity of the Linux kernel. First, extracting entries that should be analyzed is difficult because entries can be registered dynamically. In order to generate specifications for specific submodules, we need to analyze submodules' operations invoked by system calls, and we refer to these operations as *entries*. However, entries can be registered dynamically in many scenarios, for instance, during kernel initialization and module loading. Consequently, it is challenging to extract entries using current static analysis methods. Second, the input types of entries can vary in different execution paths, resulting in difficulties identifying them. The functionalities of kernel submodules are complex, while the number of entries for accessing them is limited. In consequence, the submodules' entries can accept different input types across execution paths. We need to collect input types and corresponding range constraints in each execution path of every entry, which poses significant complexity to the analysis. Finally, to generate specifications in domain languages used by kernel fuzzers, we need to perform syntax mapping and semantic encoding based on the collected information.

To address the aforementioned challenges, we propose KSG (Kernel Specification Generator) to automatically generate system call specifications for kernel fuzzers. KSG mainly contains three steps. First, in order to extract submodules' entries without being bound to their implementation details, KSG utilizes a probe-based tracing with Linux *eBPF* [2] and *kprobe* [13]. Based on the extracted entries, KSG uses path-sensitive analysis to collect precise input types and range constraints in each execution path of entries, which is based on Clang Static Analyzer (CSA) [20]. Finally, based on the gathered information, KSG generates system call specifications in domain language Syzlang, which is used by most kernel fuzzers, to improve the fuzzing efficiency. We evaluated KSG on multiple versions of Linux kernel. It generates 8 specialized calls per minute, with a total of 2433 specialized calls generated in 5 hours, 1460 of which are new to existing specifications. Leveraging the generated specifications, Syzkaller and Moonshine's [23] coverage are improved by 22% and 23%, respectively. Furthermore, KSG assisted fuzzers to discover 26 previously unknown bugs, with 13 and 6 were fixed and assigned with CVEs, respectively.

Overall, we make the following technical contributions:

- We propose an analysis approach for system call specification generation. It incorporates multiple techniques to precisely collect submodules' entries and their types and range constraints on each execution path.
- We designed and implemented KSG, which extracts submodules' entries with *eBPF* and *kprobe* and performs path-sensitive analysis based on Clang Static Analyzer. Leveraging the collected information, KSG generates specifications in Syzlang for kernel fuzzers.
- The evaluation result shows that KSG generated 2433 specifications in total, which can improve the coverage of Syzkaller and Moonshine by 22% and 23% respectively, and assisted fuzzers to find 26 new bugs.

2 Background and Related Works

2.1 Kernel Fuzzing

Fuzz testing is an automated vulnerability discovery approach. Its idea is to continuously generate input to trigger program crashes with the assistance of various sanitizers [27, 28]. Within each fuzz loop, the fuzzer selects the seed from the given corpus with certain guided strategies [5, 25]. Then, it mutates the seed by combining multiple mutation operators, and feeds the generated input to the target program for execution [7, 16]. Meanwhile, the fuzzer collects interesting program behavior, e.g., coverage [3], to determine whether an input is valuable for further mutations, thus continuously optimizing the fuzzing campaign. Take AFL [15] for instance. It is a coverage-guided userspace program fuzzer that has found hundreds of vulnerabilities in widely-used libraries. Many works optimize each part of the fuzz loop [1, 17, 18, 36, 40]. For instance, RIFF [37] moves computations done originally at runtime to instrumentation time, thus reducing the instrumentation code while utilizing vector instructions to improve throughput. Consequently, its coverage measurement mechanism can reduce fuzzing overhead significantly.

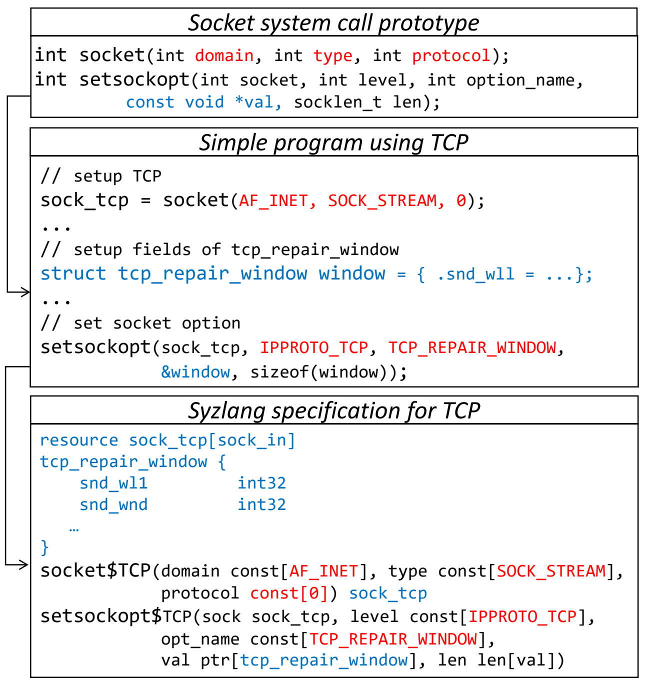
The overall process used in kernel fuzzing is similar to that used in userspace, but each specific part can be different due to the complexity of the kernel. Specifically, the idea of kernel fuzzing is to generate high-quality input to trigger kernel crash assisted with kinds of kernel sanitizers [8, 9], which is the same as a standard fuzz loop. However, unlike userspace fuzzing, the input structure of system calls can be complicated, and the cost of each test case execution is expensive.Therefore, the inputs generated by kernel fuzzers need to satisfy the structural and range constraints; otherwise, it would be rejected early by input validation, thus wasting huge amounts of fuzzing time. Existing fuzzers use specifications to encode input information of system calls to address this. Based on this domain knowledge, the performance of kernel fuzzers can be improved considerably.

Syzkaller is a state-of-the-art kernel fuzzer developed by Google. In order to generate high-quality input, it utilizes the domain language Syzlang to encode system call specifications manually. Although the encoding process brings substantial costs, they enable Syzkaller to discover thousands of bugs in the Linux kernel. Meanwhile, many works improve each part of kernel fuzzing [12, 14, 21, 26, 29, 35, 38]. Take Moonshine [23] as an example, it proposes a seed distillation algorithm to collect system call sequences from real-world applications and provide initial seeds for kernel fuzzers. Healer [30] optimizes input synthesis with system call influence relations and utilizes a dynamic learning algorithm to identify such relation between calls. Although both works improve the fuzzing performance significantly, their performance, like Syzkaller, depends on the quality and abundance of Syzlang specifications.

2.2 System Call Specification

Many system calls in Linux are an abstraction over corresponding functionalities of kernel submodules, and they are responsible for dispatching the user input to submodules' operations. A system call can accept parameters with different types based on submodules' expectations. As shown in Figure 1, the input types of socket-related calls can vary for different underlying protocols. The type of parameter `val` in `setsockopt` is `void*`, where it becomes `struct tcp_repair_window` when protocol is TCP, while other protocols can define different types. The structure of `val` can be very complex since each protocol supported by the Linux kernel can utilize different types. Besides, the original type of `val` (`void*`) does not provide any structural information to fuzzer. Without further input information, fuzzer cannot test `setsockopt` effectively since most generated inputs do not satisfy the requirements of specific protocols and will be rejected by input validation.

Kernel fuzzers utilize specifications written in domain language to generate input. For instance, Syzkaller utilizes Syzlang to encode specifications for specific submodules. Within each submodule's specification, kernel experts first define the `resource type` corresponding to the submodule. The resource type in Syzlang infers that the value of a parameter can only be constructed by the kernel and represents a kind of kernel resource. Then, kernel experts specialize system calls



Socket system call prototype

```
int socket(int domain, int type, int protocol);
int setsockopt(int socket, int level, int option_name,
        const void *val, socklen_t len);
```

Simple program using TCP

```
// setup TCP
sock_tcp = socket(AF_INET, SOCK_STREAM, 0);
...
// setup fields of tcp_repair_window
struct tcp_repair_window window = { .snd_wl1 = ...};
...
// set socket option
setsockopt(sock_tcp, IPPROTO_TCP, TCP_REPAIR_WINDOW,
           &window, sizeof(window));
```

Syzlang specification for TCP

```
resource sock_tcp[sock_in]
tcp_repair_window {
    snd_wl1         int32
    snd_wnd         int32
   …
}
socket$TCP(domain const[AF_INET], type const[SOCK_STREAM],
           protocol const[0]) sock_tcp
setsockopt$TCP(sock sock_tcp, level const[IPPROTO_TCP],
           opt_name const[TCP_REPAIR_WINDOW],
           val ptr[tcp_repair_window], len len[val])
```

Figure 1: The input types of socket-related calls can vary for different protocols. The type of parameter `val` is `void*`, it becomes `struct tcp_repair_window` when the protocol is TCP, and other protocols can define different types. With Syzlang (the bottom part), calls can be specialized to specific protocol with range constraints (highlighted in red) and precise types (highlighted in blue).

that can access the submodule to multiple simplified calls via adding range constraints and qualifying the input type. Take Figure 1 as an example, it demonstrates parts of specifications written for TCP. The resource type `sock_tcp` represents a created TCP socket. Each parameter of the specialized call `socket$TCP` is qualified as a constant value (highlighted in red), which guides kernel fuzzers to set up TCP socket correctly. The parameter `val` of `setsockopt$TCP` is qualified as `tcp_repair_window`, which is the correct type corresponding to `TCP_REPAIR_WINDOW` option. Using Syzlang, these socket-related calls can be specialized to specific protocols with range constraints and precise types. Kernel fuzzers can use specialized calls to considerably improve their efficiency.

However, manually encoding specifications can be time-consuming due to the required domain knowledge mentioned in Section 1. Several works propose to generate specifications for specific system calls or particular scenarios. DIFUZE [6] is dedicated to generating specifications for system call `ioctl` of Android drivers and is the most relevant work to ours. It first finds all uses of `file_operations` related structures to identify the handle of `ioctl`. DIFUZE then tries to extract the device name from specific registration functions in the kernel. Finally, it detects the command values and corresponding parameter structures with LLVM's analysis capabilities, e.g., range analysis. With the above steps, DIFUZE can generate correct usages of `ioctl` from different drivers. However, most submodules' operations are registered dynamically with unpredictable manners, which results in false negatives in DIFUZE. Besides, its pattern-based method can only be used to analyze `ioctl`. Meanwhile, Syzgen [4] and IMF [11] propose to generate specifications for close-sourced components of macOS. They capture system calls issued by userspace applications and generate specifications by analyzing the parameters' value of captured calls. Nevertheless, both approaches do not utilize the available source code in open source scenarios to generate more effective system call specifications.

3 Challenges

3.1 Extracting Entries of Submodules

In order to generate specifications for specific submodules, we need to analyze the submodules' entries that are invoked by system calls. Specifically, Linux defines the operations that submodules should implement with structures containing function pointers, e.g., `file_operations` and `proto_ops` as shown in Figure 2. Submodules implement these operations and register them to the kernel. We refer to these operations as entries. The responsibility of many system calls is to dispatch the input to the registered operations via indirect function call; thus, they do not contain much input information of the specific submodules. The submodules' entries define the input types to the system calls for accessing themselves. Therefore, we need to analyze specific entries to obtain concrete input types to generate high-quality specifications. Take Figure 1 as an example, `val`'s type is `void*` and the system call `setsockopt` does not make any restriction on its concrete type. In TCP scenarios, `setsockopt` passes `val` to the entry `tcp_setsockopt`, which requires `val`'s type should to be `struct tcp_repair_window*` when the option is `TCP_REPAIR_WINDOW`. We need to analyze the entry `tcp_setsockopt` to generate specifications for `setsockopt` in TCP scenarios.

However, the submodules' entries can be registered dynamically in many situations, making it difficult to extract them. Submodules implement operations and store the function pointers in the corresponding structures, which are registered to the kernel with kinds of registration functions. The process mentioned above occurs at various times, such as kernel initialization and module loading. However, identifying the pointer's target is challenging with current static analysis approach. The various registration points further increase the engineering efforts. For example, Figure 2 shows the definitions of `file_operations` and `proto_ops`, which contain the operations for kinds of files and sockets. Device drivers can implement `file_operations` according to their needs and register the structure to `VFS` during module loading.

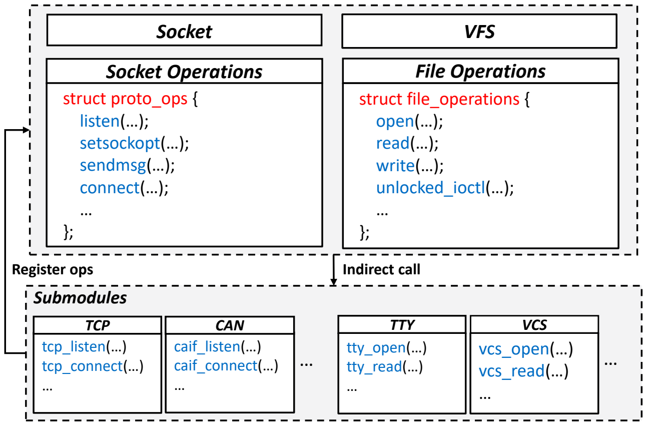


Figure 2: The definitions of `file_operations` and `proto_ops` contain the operations for files and sockets. Device drivers can implement `file_operations` as their needs and register this structure to `VFS`. Different protocols can register their own operations to socket layer in different ways.

Different protocols can register their `proto_ops` operations to the socket layer in different ways during kernel initializing. In order to generate specifications for submodules, we need to extract their entries and address the aforementioned dynamism.

3.2 Identifying Input Types of Entries

The second challenge is that each entry's input type can be different across execution paths, which further increases the complexity of the analysis. As mentioned above, the kernel defines the operations that submodules need to implement via various structures containing function pointers. The number of such function pointers in each specific structure is limited, while each submodule can be very complex. Consequently, many submodules' entries accept different input types in different execution paths to satisfy their functional requirements. In other words, the input type to the submodule's entry is not fixed; some of the input parameters are responsible for controlling the execution path, while other parameters or fields have different types depending on the value of the former. Figure 3 shows the TCP submodule's implementation of system call `setsockopt`, which is registered with `proto_ops` structure. The value of the parameter `optname` is mainly used to determine the different execution paths, while `optval` has different types based on the value of the former. In order to generate specifications, we need to identify the parameters' type and collect corresponding range constraints in each execution path of the entries.

However, variables can be aliased with each other and cast to different types by different means, resulting in the difficulty in identifying their types and collecting corresponding range constraints in each execution path. To demonstrate the former

```
static int do_tcp_setsockopt(struct sock *sk, int level,
         int optname, sockptr_t optval, unsigned int optlen)
{
    struct tcp_sock *tp = tcp_sk(sk);
    ...
    switch (optname) {
        case TCP_CONGESTION: {
            char name[TCP_CA_NAME_MAX];
Path1: ➡    // type of `optval` is char[TCP_CA_NAME_MAX]
            strncpy_from_sockptr(name, optval,  …);
        }
        case TCP_MAXSEG:
            int val;
Path2: ➡    // type of `optval` is int*
            copy_from_sockptr(&val, optval, sizeof(val));
            tp->rx_opt.user_mss = val;
        case TCP_REPAIR_WINDOW:
            struct tcp_repair_window opt;
Path3: ➡    // type of `optval` is tcp_repair_window*
            if (copy_from_sockptr(&opt, optval, sizeof(opt)))
                return -EFAULT;
        }
    return err;
}
```

Figure 3: do_tcp_setsockopt is TCP's implementation of proto_ops. The type of parameter optval varies for different value of optname. This demonstrates that the input type of submodule's entry can be different across execution paths

case, the value of variable p0 with scalar type can be assigned to another variable p1. Meanwhile, p1 can be cast to a pointer, which infers that p0 represents an address. We will miss this kind of information without handling the alias between variables. To demonstrate the latter case, as shown in Figure 3, although parameter optval is declared as sockptr_t type, it is converted to different types under different cases of switch statement using different cast methods. For instance, optval is cast to void* type with C-style cast expression before the switch statement. optval is treated as int* type on *Path 2*, because copy_from_sockptr calls copy_from_user to copy sizeof(val) bytes from userspace, while variable val is int type. The above pattern is common in the kernel, thus we need to adequately handle the aliasing issue and type casting to properly collect types and ranges constraints.

4 Key Techniques

Figure 4 shows the overall workflow of KSG. First, the kernel source code is compiled based on the given configuration, which outputs a bootable kernel image and a series of files containing the Clang AST. The AST provides the kernel with code information for each stage of the analysis. When the kernel boots, the entry extraction module hooks multiple probes dynamically before and after specific kernel functions. KSG then scans various device files and network protocols, thus triggering the execution of hooked kernel functions, which can be captured by the probes. Consequently, the probes can detect and extract the submodules' entries. Based on the AST and entries, KSG analyzes the range constraints and input types in

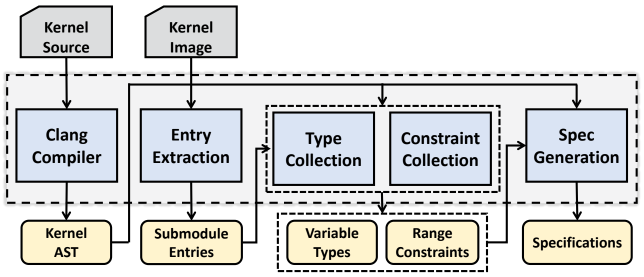


Figure 4: Workflow of KSG. The kernel code is compiled to a bootable image and files containing the clang AST. Submodules' entries can be detected and extracted by the entry extraction module. Based on the AST and entries, KSG collects the range constraints and input types in each execution path of each entry with path-sensitive analysis. Finally, KSG generates specifications based on the collected information.

each execution path of each entry with path-sensitive analysis. Finally, based on the collected information, KSG generates specifications in domain language Syzlang for fuzzers, where the syntax mapping and semantics encoding are performed. The specifications can be generated with the aforementioned process, and the effectiveness of fuzzers can be improved with the generated specifications.

4.1 Entry Extraction

As mentioned above, we need to analyze the submodules' entries for specification generation. However, the entries can be registered in many scenarios, causing difficulties in locating them. To address this, KSG utilizes a probe-based tracing to extract the entries. Although entries of different submodules can be registered with unpredictable manners, they are eventually stored into the specific data structures' fields in the kernel. For instance, entry file_operations is stored into the f_ops field of struct file, which is maintained by virtual file system (VFS). In another instance, different protocols of the net subsystem store entry proto_ops into field ops of struct socket. Therefore, instead of analyzing the entries' registration points, KSG extracts entries by capturing data structures containing the respective submodules' entries, which we refer to as target structures.

Figure 5 shows the workflow of entry extraction. When the kernel boots, KSG hooks multiple probes before and after specific kernel functions utilizing Linux *eBPF* and *kprobe* (①). eBPF and kprobe enable KSG to hook our custom functions into any kernel function. We refer to these extended functions as probes and to hooked kernel functions as target functions. The target functions we choose are a mandatory part to access the submodules and are responsible for constructing target structures, thus they enable the probes to capture the execution of them and access target structures. Then, KSG scans the kernel resources corresponding to submodules from userspace.

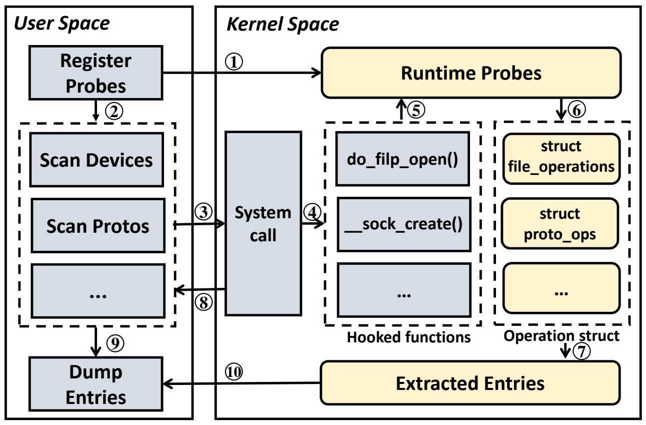


Figure 5: Workflow of entry extraction. KSG hooks probes before and after the target kernel functions. Then, it scans resources and traps into kernel space. The target functions are executed, and the whole process is captured by the probes, which extract submodules' entries and save their addresses. Finally, KSG symbolizes addresses of the entries in userspace.

For instance, KSG opens device files and specific network protocols via system calls `open` and `socket` to access *VFS* and *net* submodules (② & ③). After trapping into kernel space, these system calls invoke the target functions, which construct the target structures (④). Then the whole process is captured by the probes, which extract the submodules' entries via accessing certain field of the target structures (⑤ & ⑥). The entries are then stored into data structures provided by eBPF so that the extracted entries can be accessed from userspace (⑦ & ⑧). Finally, KSG reads the entries via *bpf* system call, and symbolize entries to the corresponding kernel symbols with Linux */proc/kallsyms*. The above process enables KSG to extract submodules' entries accurately.

Take device drivers for instance, the probes will be hooked after the execution of kernel function `do_filp_open`, which is called by system call `open` and is responsible for opening files used by `VFS`. Then, KSG accesses files in specific directories of the system recursively, e.g., `/dev` and `/proc`. After capturing the execution of `do_filp_open`, the probe first filters the threads, thus ensuring that only KSG's execution is captured. The probe accesses the kernel data `struct file`, which represents the state of an opened file, and reads the field `f_ops` of it. Field `f_ops` is `filer_operations` type and contains the submodule's entries. KSG saves `f_ops` into eBPF maps so that entries can be read and symbolized in userspace. Since Linux treats almost everything as a file and most submodules are accessible from VFS, the above procedure can extract most submodules' entries. For sockets, KSG scans all the protocols supported by the kernel and captures the kernel function `__sock_create`, which is called by system call `socket`. KSG extracts the entry `proto_ops` of each protocol by accessing the field `ops` in the `struct socket`.

4.2 Types and Constraints Collection

With submodules' entries being extracted, KSG needs to collect input information from them for specification generation. However, the parameters' types of each entry can vary across execution paths. To identify parameters' types of each execution path, KSG needs to check if a parameter, originally declared in scalar type, is cast to pointer, and collect the most precise type of each pointer. Overall, KSG utilizes the symbolic execution of Clang Static Analyzer (CSA) to perform intra-procedural, path-sensitive analysis on submodules' entries. During symbolic execution, KSG checks all expressions that can determine the parameters' types of each execution path and associate the most precise type with each parameter using the comparison rules in Table 1. When the symbolic execution of a path is finished, all the needed type information is collected and the range constraints are recorded in the CSA.

To correctly identify parameters' types, KSG first needs to handle the alias between variables. CSA associates variables with unique symbolic values and allocates a memory region for each variable based on its memory model [39]. Aliasing issue can be handled with this mechanism because CSA guarantees that variables that are aliased with each other either have the same symbol or point to the same memory region during symbolic execution. Specifically, if the symbolic value of a variable is `sym0`, then the symbolic value of variables that are assigned with the former will also be `sym0`. Variables with pointer type that have the same address during concrete execution always point to the same memory region during symbolic execution. CSA itself associates the gathered range constraints to symbolic value instead of particular variables. Since symbolic value is associated with variables and is updated accordingly during symbolic execution, range constraints can be collected and propagated by CSA.

Based on the mechanism mentioned above, we associate the type information with symbols and memory regions to collect and propagate them properly. Specifically, for variables that are originally declared in scalar type but are cast to pointers, KSG maps the symbolic value of these variables to the memory regions of pointers in `SymRegionMap` (Line 1) as shown in Algorithm 1. For pointers, KSG associates the most precise type that is known with the best effort in specific program point with their memory regions, which is stored in `RegionTypeMap` (Line 2). A special map `RegionMap` (Line 3) is used to record the connections between regions in a pointer to pointer cast. `RegionMap` is needed because CSA creates new element region for this kind of cast, while these regions represent the same variable semantically. The above three global maps can be used to record and propagate the collected type information during symbolic execution.

KSG collects input types in each execution path during CSA's symbolic execution procedure. Specifically, whenever CSA executes the type cast expression, including explicit C-style casts and implicit casts, and kernel functions with copy

Algorithm 1: Collecting Types

```
 1  $SymRegionMap := \emptyset$
 2  $RegionTypeMap := \emptyset$
 3  $RegionMap := \emptyset$
 4  for $CastExpr \in Entry$ do
 5      $S := SourceSym(CastExpr)$
 6      $T := TargetSym(CastExpr)$
 7      if $IsIntegerToPtr(CastExpr)$ then
 8          $R := Region(T)$
 9          $SymRegionMap[S] := R$
10          continue
11      if $!IsPtrToPtr(CastExpr)$ then
12          continue
13      $R0 := Region(S)$
14      $R1 := Region(T)$
15      $Record(R0, R1, RegionMap)$
16      $STy := KnownType(R0, RegionTypeMap)$
17      $TTy := KnownType(R1, RegionTypeMap)$
18      if $IsMorePrecise(STy, TTy)$ then
19          $updateRegionType(R1, STy)$
20      else
21          $updateRegionType(R0, TTy)$
```

semantics, such as copy_from_user, KSG obtains the type information from the expressions and updates the global maps mentioned above based on the comparison rules shown in Table 1. As shown in Algorithm 1, KSG records the mapping between the symbolic value of the scalar and the memory region of the pointer, which handles the integer to pointer casts (Lines 8 to 10). The recorded mapping can be used to retrieve the region of a pointer that is declared in scalar type. For a pointer to pointer cast, the algorithm first gets the respective regions of the source pointer and the target pointer (Lines 13 to 14), and records the connection between these regions into RegionMap (Line 15). Then it identifies the current known type of regions with RegionTypeMap, and the declared type of region is used if it has not been recorded (Lines 16 to 17). Based on the rules in Table 1, the algorithm updates the regions with the more precise type (Lines 18 to 21). For kernel functions with copy semantics, KSG utilizes a similar approach. Take copy_from_user as an example, KSG gets the current known type of the source pointer and the target pointer, and performs the type comparison first. Then, it checks if the last parameter is an unary expression sizeof, if so, KSG performs an additional comparison with the corresponding type. KSG updates the RegionTypeMap with the most precise type. Furthermore, KSG records the data flow direction of pointers based on the analyzed kernel functions. For example, copy_from_user infers the In direction and KSG associates this information with the corresponding memory region.

To associate the memory region with the most precise type in each execution path, KSG utilizes type comparisons. As shown in Table 1, the algorithm divides the types into four categories. First, void or void* is less precise than all other types because they do not encode any structural information. Scalar type that has longer bit width is more precise than another scalar type. Both scalar and compound type are less precise than pointer type, because it's a common use case in kernel to store the pointer value to scalar or pointer-sized compound type. For pointer types, the algorithm applies the above rules to the underlying type recursively. With the procedure above, KSG can identity the concrete type of each parameter and field of compound type.

Table 1: Rules for comparison between source type and target type. '$>$' represents that the source type is more precise than the target type, '$<$' is the opposite. *Size* means that the result depends on the size of comparison types. *Underlying* means comparing the underlying type recursively.

	Void	**Scalar**	**Compound**	**Ptr**
Void	=	$<$	$<$	$<$
Scalar	$>$	*Size*	$<$	$<$
Compound	$>$	$>$	*Size*	$<$
Ptr	$>$	$>$	$>$	*Underlying*

Figure 6 shows a running example of do_tcp_setsockopt with Algorithm 1. First, CSA marks the input parameter optname and optval as symbol sym0 and sym1, respectively. After the first case condition, it collects range constraint of sym0, indicating that optname equals to TCP_REPAIR_OPTIONS on the current execution path. Algorithm 1 is invoked when CSA enters the kernel function copy_from_sockptr since it calls copy_from_user eventually. Symbolic value sym1 is associated with memory region region0, which is recorded in SymRegionMap, because optval (integer type) is cast to a pointer. The mapping from region0 to struct tcp_repair_opt is also recorded in RegionTypeMap. Finally, CSA captures another range constraint of opt's field opt_code. In this way, KSG knows the concrete type of sym1 based on the information in SymRegionMap and RegionTypeMap. This example demonstrates that the types and range constraints can be collected properly and the second challenge can be addressed by combining CSA and type collection.

4.3 Specification Generation

Leveraging the above approach, KSG can collect parameters' types and range constraints in each execution path of submodules' entries. Based on the collected information, KSG generates specifications in domain language Syzlang for kernel fuzzers. The generation procedure needs to accomplish two

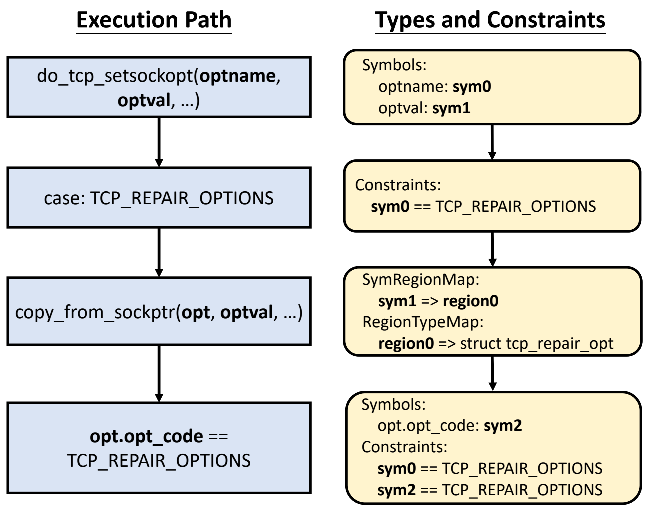

Execution Path
Types and Constraints
do_tcp_setsockopt(optname, optval, ...)
case: TCP_REPAIR_OPTIONS
copy_from_sockptr(opt, optval, ...)
opt.opt_code == TCP_REPAIR_OPTIONS
Symbols:
optname: sym0
optval: sym1
Constraints:
sym0 == TCP_REPAIR_OPTIONS
SymRegionMap:
sym1 => region0
RegionTypeMap:
region0 => struct tcp_repair_opt
Symbols:
opt.opt_code: sym2
Constraints:
sym0 == TCP_REPAIR_OPTIONS
sym2 == TCP_REPAIR_OPTIONS

Figure 6: Running example for Algorithm 1. CSA first marks `optname` and `optval` as symbolic value `sym0` and `sym1`. Then it captures range constraint on symbolic value `sym0`. Algorithm 1 maps `sym1` to memory region `region0` since `optval` is cast to pointer type. Finally, CSA further captures another range constraint of `sym2`, symbolic value of `opt`'s field `opt_code`.

major goals: syntax mapping and semantic encoding. The former performs the mapping from C language AST to Syzlang AST and the latter encodes the collected range constraints into the generated specifications.

KSG divides the generation process into two steps. The first step generates the definitions of Syzlang resource type corresponding to the submodule, and the system calls that are responsible for creating the former. As mentioned in Section 4.1, KSG scans device files and protocols to extract submodules' entries. Meanwhile, the needed information for accessing the submodule is recorded. For instance, the file paths are saved for device drivers and the `domain`, `type` and `proto` are saved for specific sockets. Based on this information, KSG defines *resource* type for each submodule, and the name of the defined *resource* type follows specific rules. For example, resource types for device and socket are prefixed with `fd` and `sock`, respectively. For TCP submodule shown in Figure 1, `sock_tcp` is defined in this step. Then, KSG generates the corresponding system calls that create the resource type. For example, KSG generates the system call `open` for device drivers, and the input path of `open` is qualified to the file path of the device. The specialized version of system call `socket` is generated for each protocol, e.g., `socket$TCP` shown in Figure 1.

The second step generates the specialized calls for the remaining entries of a submodule. Specifically, KSG generates a specialized call for each execution path of each entry. The duplicated calls are filtered, and the parameters' types of each generated specialized call are qualified to the corresponding type in the execution path. Specifically, for a variable declared in scalar type, KSG first checks if it is a pointer via querying `SymRegionMap` and maps it to Syzlang pointer if so; otherwise, KSG obtains its bit size according to the AST information and maps it to the corresponding numeric type with same bit size in Syzlang. Meanwhile, KSG checks whether the symbol of the scalar has range constraints by querying the program state of CSA. The corresponding constraint is represented as Syzlang's `const` type or ranged integer type. For array type, KSG first maps its element type to Syzlang type recursively, then queries CSA whether its length has range constraints. KSG constructs the corresponding Syzlang array type based on the mapped element type and length information. For pointer type, KSG first gets the memory region of the pointer from CSA, and queries the concrete type associated with the region from `RegionTypeMap`. KSG then maps the type of pointee recursively, and queries data flow direction associated with the memory region. KSG constructs the Syzlang pointer type with the mapped pointee's type and collected data flow direction. Finally, KSG maps each field of compound type to Syzlang type and generates the corresponding compound type in Syzlang. Based on the above mapping rules, KSG can generate specifications for submodules' entries based on the collected types and range constraints.

Take Figure 1 as an example, KSG generates three specialized system calls for each path of `do_tcp_setsockopt`. The type of `optname` is mapped to `const` type in Syzlang based on range constraints of each path. In the meantime, the type of `optval` is mapped to `array`, `int32`, and `struct`, respectively. Listing 1 in the Appendix shows part of generated specifications for driver `/dev/pts`, which manual specifications do not cover. Listing 2 shows part of generated specifications for the socket `X25`.

5 Implementation

Entry extraction. We implement eBPF programs based on BCC [10] and hook them as kprobe into target kernel functions. Two probes are used to capture the entries of device drivers' and protocols' operations, respectively. We currently utilize a userspace program to trigger the extraction process. The program first attaches the probes to the kernel. It then scans kernel-provided resources, such as opening files in `/dev`, mounting all the supported file systems, opening files in different file systems, and creating all the supported sockets of the kernel. These operations allow us to extract the implementation of the corresponding file operations and socket operations for different submodules.

Types and Constraints. We implement the types and constraints collection based on Clang13. Algorithm 1 is implemented as multiple CSA checkers that are hooked after each time CSA simulates execution of cast expression and before the execution of functions with copy semantics, e.g., `copy_from_user`. These checkers read the symbol values and memory regions of the expressions in the hooked operations from current program state, and update the type information stored in the global maps based on the type comparison rules in Table 1. For better intra-procedure analysis, we utilize the cross translation unit (CTU) analysis of CSA based on pre-dumped AST and compilation database. We customized the analysis configuration, e.g., increasing the max number of imported translation units, limiting the loop time since it does not provide new information for specification generation but reduces efficiency. Besides, we also modeled a larger number of kernel library APIs via implementing CSA checkers for better symbolic execution, including `kmalloc`, string manipulation functions, etc. These checkers observe the symbolic execution of the kernel and actively participate in modeling the program behavior via modifying the region bindings and range constraints stored in the program state.

Specification Generation. The generation procedure is implemented as plugins too that are hooked into CSA at the end of each execution path's simulation. We first implement AST to fully support Syzlang language. Based on the type information stored in the global maps and the range constraints of each symbol in the CSA, the translation of KSG maps the C language AST to Syzlang AST. In order to generate the specifications, the mapped AST is serialized into text format that conforms to the syntax rules of Syzlang, thus allowing kernel fuzzers to use the generated specifications and speed up the entire fuzzing campaign.

6 Evaluation

In this section, we evaluate the effectiveness of KSG on recent versions of Linux and fuzzers. Specifically, we chose Linux-5.15, 5.10, and 5.4 as our target versions. Linux 5.15 is the latest version prior to submission, whereas 5.10 and 5.4 are widely used by many distributions. To evaluate the effectiveness of the generated specifications in improving fuzzers' performance, we took the generated specifications as input to Syzkaller and Moonshine, and compared the code coverage and bug finding capabilities to their original version. We chose Syzkaller because it is the state-of-the-art kernel fuzzer. Moonshine improves Syzkaller's fuzzing efficiency by distilling high-quality seeds for it and is a representative fuzzer. We design experiments to address the following questions:

- **RQ1:** How does KSG perform in generating system call specifications in terms of efficiency and quality?
- **RQ2:** How effective are the generated specifications in improving the coverage of kernel fuzzers?
- **RQ3:** How effective are the generated specifications in assisting kernel fuzzers to find bugs?

Experiment Settings The experiments were conducted on a Linux server with a 16-core Intel i7-10700K CPU and 32 GiB of memory. Each version of the kernel uses the same compilation configuration. Specifically, `CONFIG_BPF` and `CONFIG_KPROBE` were enabled for entry extraction. We also enabled `CONFIG_KCOV` to collect code coverage. We extended fuzzers with the generated specifications, and we refer to those extended fuzzers as Syzkaller+ and Moonshine+, respectively. All 4 fuzzers were configured with the same parameters in terms of QEMU configurations and base system call specifications. Specifically, we started all experiments simultaneously and distributed the resources evenly, including 2 cores and 4 GiB of memory for each virtual machine. All 4 fuzzers adopted the same base version of the `Syzlang` specifications. To reduce statistical errors, each experiment was repeated 3 times and executed over a period of 72 hours, and the average results were reported.

6.1 Specification Generation

We executed KSG on three versions of the Linux kernel and the whole process of specification generation is automatic. Table 2 shows the results of this process. During entry extraction, KSG scanned 1098 unique device files and 78 different sockets in total, and extracted 572 and 222 entries, respectively. Note that the number of entries is not equal to the number of scanned files and sockets multiplied by the number of function pointers defined in `file_operations` and `proto_ops`. This is because: first, the registered operations of different files and sockets can be the same; second, each submodule does not need to implement all the operations; finally, KSG de-duplicates the extracted entries and verifies the extracted addresses. Besides, we manually verified the correctness of extracted entries by reading the source code corresponding to the submodule. The result shows that KSG can correctly extract the entries of all files and sockets that are accessible from userspace. Furthermore, since KSG performs entry extraction dynamically based on eBPF after kernel booted and all submodules loaded, the correctness of the extracted entries can also be guaranteed.

Table 2: KSG extracted 792 entries by scanning 78 sockets and 1098 device files. After path-sensitive analysis in 5h, KSG generated specifications containing 2433 specialized calls, and 1460 of them are new to existing specifications.

	Scanned	Entries	Specs	New Specs
Socket	78	222	923	+586
Driver	1098	572	1510	+874
Overall	1176	794	2433	**+1460**

By performing path-sensitive analysis on submodules' entries, KSG generates 8 specialized calls per minute, with a total of 2433 specialized calls generated in 5 hours. Of this total, 1510 specialized calls are generated from device drivers while 923 specialized calls corresponded to sockets. Specifically, for

64% of the extracted entries, the number of generated specialized calls is less than 2. This is because the input types remain consistent across the execution paths. For instance, KSG generates one specialized call for system call `bind` of each type of socket. The input address is qualified to the type defined by the corresponding socket type, while the type of such parameter in C prototype does not constrain the input structure. Although the number of specialized calls for this 64% of the entries is limited, encoding specifications for them requires extensive domain knowledge, whereas KSG can automate this process leveraging the source code analysis. For 36% of the extracted entries, the number of generated specialized calls is more than 2 because the input types of these entries can vary in different execution paths. The average number of specialized calls for these entries is 4, and KSG generates up to 29 specialized calls for system call `getsockopt` of `X25` socket. For those 36% of the entries, the manual efforts of writing specifications would be vast, while the automation of KSG can significantly reduce the time cost of this process.

Compared with the existing specifications that contains 1204 specialized calls for the drivers and sockets scanned by KSG, 1460 generated calls are new, of which 586 and 874 are generated from the analyzed sockets and drivers, respectively. In order to further verify the correctness of the generated specifications, we manually checked if the range constants match the collected parameter types by reading the source code of submodules. The final result shows that KSG can correctly extract the input types and the corresponding range constraints in each execution path of submodules' entries.

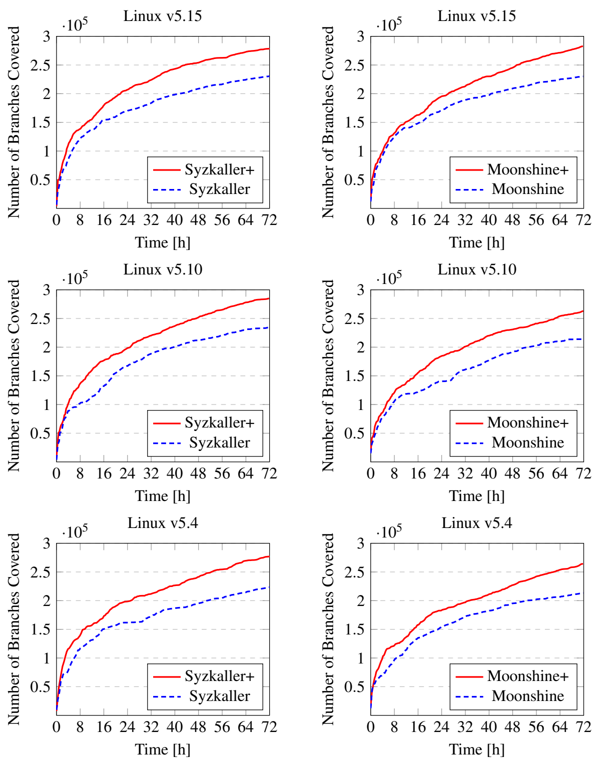


Figure 7: Coverage growth of Syzkaller and Moonshine with generated system call specifications on three versions of Linux kernel over 72 hours. In all kernel versions, Syzkaller+ and Moonshine+ achieve the higher coverage statistics.

6.2 Coverage Improvement

To answer **RQ2**, we took 1460 new specialized calls as input to Syzkaller and Moonshine, while monitoring the fuzzing process and sampling each fuzzer's statistics in the 72-hour run. Figure 7 shows the comparison of branch coverage between fuzzers and Table 3 lists detailed statistics. The base specifications used by Syzkaller and Moonshine contain 4144 specialized calls in total, including specifications encoded for submodules that have not been handled by KSG. With 1460 new specialized calls, Syzkaller+ and Moonshine+ achieved 22% and 23% coverage improvement, respectively.

As shown in Figure 7, both Syzkaller+ and Moonshine+ can achieve higher coverage statistics than their original version in the same amount of time. Specifically, all tools show significant growth in the first 8 hours, where the advantage of the generated specifications is not obvious. After fuzzing for 8 hours, the coverage growth of Syzkaller and Moonshine starts to slow down, whereas that of Syzkaller+ and Moonshine+ is significantly faster than the former. This is because KSG does not improve kernel fuzzers' throughput or efficiency, but rather enables fuzzers to reach more modules and code with additional generated specifications. Therefore, all fuzzers perform at similar rates before 8 hours since they have yet to cover the code reachable using manually-written specifications. They diverge after 8 hours as the manually-written specifications cannot provide the kernel fuzzers with more low hanging fruit, while the generated specifications allow the fuzzers to continue finding more code in more modules.

In principle, fuzzers utilize the generated specifications to generate test cases. In order to further demonstrate the reason behind the improvement, we analyzed the output corpus of all fuzzers and calculated the percentage of test cases that contain the newly generated calls in the whole corpus. At the end of 72-hour experiment, the average percentage of such inputs in the corpus is 28%. Therefore, the reason behind the coverage improvement is that the generated specifications provide new test portals for fuzzers. Syzkaller+ and Moonshine+ can synthesize test cases based on the new specifications thus covering kernel code that used to be unreachable. Meanwhile, the improvement can demonstrate the quality of the generated specifications, since specifications with low quality, e.g., range constraints mismatch input types, can even hinder fuzzers' capabilities. The above results prove that the generated specifications can assist fuzzers in exploring more code in the kernel.

Table 3: Coverage statistics of fuzzers compare to their original versions. Columns "min-impr" and "max-impr" present the minimum / maximum improvement.

(a) Syzkaller+ vs. Syskaller

Version	min-impr	max-impr	Average
5.15	+18%	+24%	+21%
5.10	+19%	+25%	+22%
5.4	+20%	+28%	+24%
Overall	+19%	+25%	**+22%**

(b) Moonshine+ vs. Moonshine

Version	min-impr	max-impr	Average
5.15	+19%	+24%	+22%
5.10	+20%	+25%	+23%
5.4	+20%	+26%	+24%
Overall	+19%	+25%	**+23%**

6.3 Bug Finding and Case Studies

To answer **RQ3**, we tested the Linux kernel with Syzkaller+ and Moonshine+ for two weeks. As a result, we found 138 unique vulnerabilities in total and 26 were confirmed by maintainers as previously unknown bugs, of which 13 and 6 were fixed and assigned with CVEs, respectively. Table 4 lists the details of those vulnerabilities. Most of these vulnerabilities are critical. For instance, KSG assisted fuzzers to discover a vulnerability with a 7.0 CVSS score (CVE-2021-4028). Although Syzkaller has been testing the Linux kernel continuously with large amounts of computing resources, these 26 vulnerabilities have not been reported. The reason why KSG assisted Syzkaller+ and Moonshine+ to discover 26 previously unknown vulnerabilities is that the generated specifications provide fuzzers with more information of system calls. Based on this domain knowledge, Syzkaller+ and Moonshine+ can generate test cases to test kernel code that used to be difficult for fuzzers to reach. The above result shows that the specifications automatically generated by KSG can improve the fuzzers' vulnerability detection capabilities.

Case Study: CVE-2021-4148 KSG assisted fuzzers to discover a vulnerability in `VFS`. As shown in Figure 8, `block_invalidatepage()` would throw `BUG` due to assertion failure if *stop* is greater than `PAGE_SIZE`. However, the input generated by the fuzzer is a huge page, and the length is the size of the huge page due to the read-only `FS THP` support. This triggers kernel crash directly and Figure 8 shows a fix for this. However, the root cause of this vulnerability is complicated. Specifically, the kernel isn't supposed to get a writable file descriptor on a file that has huge pages added to the page cache without the filesystem's knowledge. VFS should have truncated the page cache when it found `THPs` in the cache. Except for the fix mentioned, this vulnerability was fixed properly with additional patches. Leveraging the newly generated specifications, the fuzzer synthesized the corresponding test cases thus triggering the assertion failure.

Table 4: KSG assisted fuzzers to discover 26 previously unknown vulnerabilities. All of these vulnerabilities have been confirmed by maintainers; 13 of these bugs have been fixed by corresponding patches and another 6 have been assigned with CVEs.

Operation	**Risk**	**Status**
sk_stream_kill_queues	logic bug	*Fixed*
__init_work	use after free	***CVE-2021-4150***
truncate_inode_page	logic bug	*Fixed*
__folio_mark_dirty	logic bug	*Fixed*
kvm_arch_vcpu_create	logic bug	***CVE-2021-4032***
cma_cancel_listens	use after free	*Fixed*
io_wq_submit_work	logic bug	***CVE-2021-4023***
btrfs_alloc_tree_block	logic bug	*Fixed*
__btrfs_tree_lock	deadlock	***CVE-2021-4149***
smp_call_function	soft lockup	*Confirmed*
block_invalidatepage	dereference null	***CVE-2021-4148***
rdma_listen	use after free	***CVE-2021-4028***
ext4_block_write_begin	logic bug	*Confirmed*
io_ring_exit_work	task hung	*Fixed*
skb_try_coalesce	task hung	*Confirmed*
btrfs_search_slot	deadlock	*Fixed*
__set_page_dirty	logic bug	*Confirmed*
__kernel_read	logic bug	*Fixed*
xlog_cil_commit	dereference null	*Fixed*
hub_port_init	task hung	*Confirmed*
hci_cmd_timeout	logic bug	*Confirmed*
cgroup_rstat_flush_locked	data race	*Fixed*
btrfs_free_tree_block	logic bug	*Confirmed*
io_uring_cancel_generic	task hung	*Fixed*
hci_uart_tx_wakeup	logic bug	*Fixed*
blk_mq_get_tag	logic bug	*Fixed*

7 Discussion and Limitations

During the experiments, we found a total of 231 specialized calls from existing specifications that are encoded for the submodules scanned by KSG, but are nonexistent in the generated specifications. We believe there are three major reasons for this. First, KSG mainly considers range constraints while handwritten specifications encode other parameters semantics, e.g., defining parameters that are checksums of other fields as instances of the `csum` type in Syzlang. Second, kernel experts can redefine original input types from system call definitions to other types with the same memory layout to generate argument values more efficiently. For instance, some submodules use the high 16 bits and low 16 bits of a `u32` number for different purposes, and kernel experts redefine them as two `u16` types so that fuzzer can generate and mutate values for them individually. For these two limitations, we can improve KSG further by hooking more kernel functions during

```
diff --git a/fs/buffer.c b/fs/buffer.c
index ab7573d72dd7..4bcb54c4d1be 100644
--- a/fs/buffer.c
+++ b/fs/buffer.c
@@ -1507,7 +1507,7 @@ void block_invalidatepage(struct
page *page, unsigned int offset,
        /*
         * Check for overflow
         */
-       BUG_ON(stop > PAGE_SIZE || stop < length);
+       BUG_ON(stop > thp_size(page) || stop < length);

        head = page_buffers(page);
        bh = head;
@@ -1535,7 +1535,7 @@ void block_invalidatepage(struct
page *page, unsigned int offset,
         * The get_block cached value has been
unconditionally invalidated,
         * so real IO is not possible anymore.
         */
-       if (length == PAGE_SIZE)
+       if (length >= PAGE_SIZE)
                try_to_release_page(page, 0);
 out:
        return;
```

Figure 8: When the size of the `stop` is greater than `PAGE_SIZE`, `block_invalidatepage()` would throw `BUG`. Fuzzer triggered this crash by passing a huge page, where the length is the size of huge page due to `FS THP` support. This figure shows a direct fixing patch for CVE-2021-4148.

path-sensitive analysis to collect more parameters' semantics as well as redefining input types based parameters' usage. Finally, the kernel code contains low-level operations, e.g., inline assembly, which may not be well modeled by CSA, thus leading to the range constraints and type information being missed during the analysis procedure. To address this, we can construct checkers to simulate common low-level operations so that the related information can be collected and propagated properly.

Currently, we mainly apply KSG to generate specifications for drivers and sockets. Since many resources in Linux are represented as files in VFS , using file-operation-relevant system calls allows us to extract entry information for many submodules. Meanwhile, in principle, KSG is generalizable. For other multiplexing system calls, we can adapt entry extraction to the target through a slight analysis of the internal implementation to find the kernel functions that need to be injected; then, we can apply the rest of KSG. For other system calls, we can directly execute the collecting algorithm of KSG and generate specifications based on gathered information since these steps only depend on the source code information. Take system call `prctl()` as an example, KSG can collect the argument constraints directly from `sys_prctl()`.

8 Conclusion

In this paper, we propose KSG to automatically generate system call specifications for kernel fuzzers based on entry extraction and types and constraints collection. The evaluation shows that KSG generates 8 specialized calls per minute, with a total of 2433 specialized calls generated in 5 hours. Leveraging the generated specifications, Syzkaller and Moonshine's coverage were improved 22% and 23%, respectively. Furthermore, KSG assisted fuzzers to discover 26 previously unknown bugs. The above result demonstrates that KSG is effective in generating system call specifications, and the generated specifications can improve the fuzzers' performance.

For future work, we will extend KSG to other system calls or submodules to generate more specifications since some submodules are not covered by KSG yet, and submodules like drivers can change over time, which potentially involves making modifications to entry extraction. More importantly, we can augment KSG to infer semantic information of system calls' parameters, thus significantly improving the generated specifications, which can be implemented with multiple CSA checkers encoded carefully with domain knowledge.

Acknowledgments

We sincerely appreciate the guidance from our shepherd. We would also like to thank the anonymous reviewers for their valuable comments and input to improve our paper. This research is sponsored in part by the NSFC Program (No. 62022046, 92167101, U1911401, 62021002, 62192730), National Key Research and Development Project (No. 2019YFB1706200, No2021QY0604).

References

[1] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. FUDGE: Fuzz Driver Generation at Scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 975–985, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Daniel Borkmann. Linux eBPF. https://ebpf.io.

[3] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[4] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. SyzGen: Automated Generation of Syscall Specification of Closed-Source MacOS Drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 749–763, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1967–1983, Santa Clara, CA, August 2019. USENIX Association.

[6] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2123–2138, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-Based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 206–215, New York, NY, USA, 2008. Association for Computing Machinery.

[8] Google. Kernel address sanitizer. `https://www.kernel.org/doc/html/latest/dev-tools/kasan.html`.

[9] Google. Kernel concurrency sanitizer. `https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html`.

[10] Brendan Gregg'. BPF Compiler Collection. `https://www.iovisor.org/technology/bcc`.

[11] HyungSeok Han and Sang Kil Cha. IMF: Inferred Model-Based Fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2345–2358, New York, NY, USA, 2017. Association for Computing Machinery.

[12] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding Kernel Race Bugs through Fuzzing. In *IEEE Symposium on Security and Privacy*, pages 754–768. IEEE, 2019.

[13] Jim Keniston. Linux Kprobe. `https://www.kernel.org/doc/html/latest/trace/kprobes.html`.

[14] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *NDSS*, 2020.

[15] lcamtuf. American fuzzy lop, 2013. `https://lcamtuf.coredump.cx/afl/`.

[16] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 475–485, New York, NY, USA, 2018. Association for Computing Machinery.

[17] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 154–170, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.

[18] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. PAFL: Extend Fuzzing Optimizations of Single Mode to Industrial Parallel Mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 809–814, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 562–566, 2018.

[20] LLVM Developer Group. Clang Static Analyzer. `https://clang-analyzer.llvm.org/`.

[21] Dominik Maier, Benedikt Radtke, and Bastian Harren. Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing. In *Proceedings of the 13th USENIX Conference on Offensive Technologies*, WOOT'19, page 8, USA, 2019. USENIX Association.

[22] Andy Nguyen. CVE-2020-12352, 2020. `https://nvd.nist.gov/vuln/detail/CVE-2020-12352`.

[23] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, Baltimore, MD, August 2018. USENIX Association.

[24] Manfred Paul. CVE-2021-3490, 2021. `https://nvd.nist.gov/vuln/detail/CVE-2021-3490`.

[25] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing Seed Selection for Fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 861–875, San Diego, CA, August 2014. USENIX Association.

[26] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, Vancouver, BC, August 2017. USENIX Association.

[27] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[28] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, page 62–71, New York, NY, USA, 2009. Association for Computing Machinery.

[29] Yuheng Shen, Hao Sun, Yu Jiang, Heyuan Shi, Yixiao Yang, and Wanli Chang. Rtkaller: State-Aware Task Generation for RTOS Fuzzing. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.

[30] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. *HEALER: Relation Learning Guided Kernel Fuzzing*, page 344–358. Association for Computing Machinery, New York, NY, USA, 2021.

[31] Dmitry Vyukov and Andrey Konovalov. Syzbot, 2015. https://syzkaller.appspot.com/upstream.

[32] Dmitry Vyukov and Andrey Konovalov. Syzbot Dashboard, 2015. https://storage.googleapis.com/syzkaller/cover/ci-qemu-upstream.html.

[33] Dmitry Vyukov and Andrey Konovalov. Syzkaller: an unsupervised coverage-guided kernel fuzzer, 2015. https://github.com/google/syzkaller.

[34] Dmitry Vyukov and Andrey Konovalov. Syzlang: System Call Description Language, 2015. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[35] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758. USENIX Association, August 2021.

[36] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 61–64, New York, NY, USA, 2018. Association for Computing Machinery.

[37] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 147–159. USENIX Association, July 2021.

[38] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660, 2020.

[39] Zhongxing Xu, Ted Kremenek, and Jian Zhang. A memory model for static analysis of c programs. In *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I*, ISoLA'10, page 535–548, Berlin, Heidelberg, 2010. Springer-Verlag.

[40] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. Intelligen: automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 318–327. IEEE, 2021.

9 APPENDIX

9.1 Generated Specifications

Listing 1 shows part of generated system call specifications for /dev/pts. During entry extraction, KSG extracts struct file_operation of device /dev/pts via accessing device files under /dev/pts directory. KSG then performs path sensitive analysis on each extracted entry to collect types and range constraints. During the first step of generation, KSG defines the resource type fd_dev_pts_0, where the name of resource type is generated based the accessed file. openat$dev_pts_0_0 is also generated during this step and the target file path is qualified to /dev/pts/0. After the second step of the generation, the rest of specialized calls and related types were generated based on the collected types and range constraints.

Listing 1: Generated specifications for /dev/pts driver

```
resource fd_dev_pts_0[fd]
openat$dev_pts_0_0(fd const[AT_FDCWD], file
    ptr[in, string["/dev/pts/0"]], flags flags
    [open_flags], mode flags[open_mode])
    fd_dev_pts_0
...
ioctl$dev_pts_0_16(fd fd_dev_pts_0, cmd const
    [0x5413], arg ptr[in, winsize])
ioctl$dev_pts_0_11(fd fd_dev_pts_0, cmd const
    [0x80045440], arg ptr[in, int32])
ioctl$dev_pts_0_3(fd fd_dev_pts_0, cmd const[0
    x541f], arg ptr[in, serial_struct])
ioctl$dev_pts_0_5(fd fd_dev_pts_0, cmd const[0
    x545d], arg ptr[in, serial_icounter_struct
    ])
...

serial_icounter_struct {
    cts int32
    dsr int32
    rng int32
    dcd int32
    rx int32
    tx int32
    frame int32
    overrun int32
    parity int32
    brk int32
    buf_overrun int32
    reserved array[int32, 9]
}
serial_struct {
    ...
    closing_wait2 int16
    iomem_base ptr[out, array[int8]]
    ...
}
winsize {
    ws_row int16
    ws_col int16
    ws_xpixel int16
    ws_ypixel int16
}
```

Listing 2 shows part of generated system call specifications for socket X25. During entry extraction, KSG extracts struct proto_ops of X25 via invoking system call socket with address family AF_X25. KSG then performs path sensitive analysis on each extracted entry to collect types and range constraints. During the first step of generation, KSG defines the resource type sock_X25_SeqPacket, where the name of resource type is generated based the address family and socket type. socket$X25_SeqPacket is also generated during this step and the parameters are qualified to corresponding constant. After the second step of the generation, the rest of specialized calls and related types were generated based on the collected types and range constraints.

Listing 2: Generated specifications for X25 socket

```
resource sock_X25_SeqPacket[sock]

socket$X25_SeqPacket(domain const[0x9], type
    const[0x5], proto const[0x0])
    sock_X25_SeqPacket
bind$X25_SeqPacket_0(sock sock_X25_SeqPacket,
    addr ptr[in, sockaddr_x25], len bytesize[
    addr])
...
setsockopt$X25_SeqPacket_0(sock
    sock_X25_SeqPacket, level const[0x106],
    opt_name const[0x1], buf ptr[in, int32],
    len ptr[in, int32])
...
ioctl$X25_SeqPacket_6(fd sock_X25_SeqPacket,
    cmd const[0x89e5], arg ptr[in,
    x25_calluserdata])
ioctl$X25_SeqPacket_4(fd sock_X25_SeqPacket,
    cmd const[0x89ec], arg ptr[in,
    x25_causediag])
ioctl$X25_SeqPacket_9(fd sock_X25_SeqPacket,
    cmd const[0x89ea], arg ptr[in,
    x25_dte_facilities])
ioctl$X25_SeqPacket_10(fd sock_X25_SeqPacket,
    cmd const[0x89e3], arg ptr[in,
    x25_facilities])
...
sockaddr_x25{
    sx25_family const[0x9, int16]
    sx25_addr x25_address
}
x25_address{
    x25_addr array[int8, 16]
}
x25_calluserdata {
    cudlength int32
    cuddata array[int8, 128]
}
x25_causediag {
    cause int8
    diagnostic int8
}
x25_dte_facilities {
    ...
}
x25_facilities {
    ...
}
```