SyzGen++: Dependency Inference for Augmenting Kernel Driver Fuzzing

Weiteng Chen*, Yu Hao†, Zheng Zhang†, Xiaochen Zou†,
Dhilung Kirat‡, Shachee Mishra‡, Douglas Schales‡, Jiyong Jang‡, and Zhiyun Qian†
* *Microsoft Research, Redmond,* † *University of California, Riverside,* ‡ *IBM Research, Yorktown Heights*
* *weitengchen@microsoft.com,* † *{yhao016, zzhan173, xzou017}@ucr.edu, zhiyunq@cs.ucr.edu,*
† *{dkirat, schales, jjang}@us.ibm.com, shachee.mishra@ibm.com*

***Abstract*—In recent years, kernel fuzzing research has experienced a significant surge. Among various kernel fuzzers, Syzkaller stands out as the state-of-the-art tool, having identified over 5,000 bugs in the Linux kernel. Syzkaller's success can be attributed to its utilization of manually-curated syscall specifications provided by kernel experts. However, this process is time-consuming and not scalable due to complex input structures and unknown dependencies among syscalls. Consequently, a substantial portion of the kernel codebase, specifically kernel drivers, lacks specifications, posing a significant security risk.**

In this paper, we introduce SyzGen++, an innovative approach for automatically inferring dependencies between syscalls and generating specifications without relying on existing test suites. Specifically, we define two fundamental building blocks of *insertion* and *lookup* operations and their pairing to accurately identify dependencies. We evaluated SyzGen++ against existing state-of-the-art techniques on both Linux and macOS drivers. Our results demonstrate that SyzGen++ uncovered 245 more dependencies. Furthermore, SyzGen++ outperforms DIFUZE, KSG, and SyzDescribe in terms of code coverage, achieving 71%, 67%, and 39% improvement on average, respectively. Notably, our evaluation discovered 10 previously unknown bugs in Linux Kernel 6.2 using specifications generated by SyzGen++, resulting in 6 CVEs, which demonstrates its effectiveness in identifying vulnerabilities.

***Index Terms*—Fuzzing, Operating System Security, Vulnerability Analysis**

1. Introduction

The monolithic nature of the kernel, without a clear separation between the core kernel and drivers, results in driver bugs causing severe consequences (*e.g.,* privilege escalation). According to a Google report [1] and prior research [2], [3], the majority of kernel bugs are attributed to drivers, as they constitute a large portion of the codebase and often lack proper vetting.

Syzbot [4], Google's continuous kernel fuzz testing platform, represents the state-of-the-art and has discovered 5,667 bugs at the time of writing, showcasing the effectiveness of its underlying kernel fuzzer, Syzkaller [5]. The essential ingredient for its success is the syscall specifications handcrafted by kernel experts. These specifications encode both the structures and constraints of syscall arguments (*e.g.,* type and value ranges) and dependencies between syscalls. Since a kernel module maintains its internal state, successful execution of syscalls usually requires a valid sequence of invocation (i.e., *implicit dependency* or *ordering dependency*) and/or correctly passing a "handler" (*e.g.,* file descriptor) returned from one syscall to another (*i.e., explicit dependency* or *value dependency*) [6].

Although Syzkaller does not allow users to specify implicit dependencies, it can gradually learn the correct invocation order of any pair of syscalls based on their frequencies of occurrence in the corpus. For instance, a successful call to the 'recvmsg' syscall on a socket can only occur if there is data available, which can be ensured by calling 'sendmsg' beforehand. Even though the former does not require any input from the output of the latter, Syzkaller may still assign a higher probability to this pairing as it frequently leads to new coverage, and hence, new seed programs. In contrast, missing explicit dependencies in the specification is more detrimental, as it is extremely unlikely for a fuzzer to generate a random value that matches a specific "handler" returned by previous syscalls, resulting in only exploring shallow paths. For example, without knowing the explicit dependency between syscalls `open` and `read`, it is difficult to successfully invoke `read` by randomly generating an integer for its first argument (*i.e.,* file descriptor).

Developing syscall specifications is a tedious and error-prone process (as demonstrated in our evaluation §5). Despite the substantial attack surface introduced by drivers, the majority of them lack readily available specifications. In fact, Syzkaller contains only a limited number of driver specifications: 60 for Linux, 0 for Darwin, and 5 each for FreeBSD and Android (excluding non-Android specific drivers). To address this issue, an attractive direction is to automate the process of specification generation and explicit dependency inference. DIFUZE [7] analyzes Linux and Android kernel source code to extract syscall specifications, while KSG [8] uses Clang Static Analyzer to collect constraints of syscall arguments for Linux kernel. These approaches cannot be applied to closed-source kernel

**This work was done when the author was a student at UC, Riverside.*

drivers such as those in Windows and macOS (although macOS has a limited number of open-source drivers [9]). Furthermore, many Android OEM vendors selectively open source their kernels and only periodically release source code snapshots with significant delays [10], [11]. Regarding dependency analysis, IMF [6] and Moonshine [12] infer dependencies using traces produced by existing test suites, but it is challenging to collect traces for kernel drivers. IMF and Moonshine primarily target core APIs/syscalls, which have many existing applications/test suites. In contrast, drivers have relatively fewer test suites. In our evaluation (see §5 for details), Moonshine's traces covered merely five drivers, while SyzGen [9] managed to obtain traces for only about 10% of driver interfaces.

In this paper, we introduce SyzGen++, the *first framework that enables the inference of explicit dependencies without requiring source code or execution traces.* The key insight of SyzGen++ is based on the following observations:

1) In user space, an explicit dependency involves a integer (or handler) that corresponds to a complex object maintained by the kernel, *e.g.,* a file descriptor corresponding to a `struct file` in kernel space;
2) In kernel space, an explicit dependency requires a producer that creates an object and performs an *insertion* into some global data container (*e.g.,* array or linked list) and a consumer that retrieves the corresponding internal object by performing a *lookup* into the same global data container based on the user input.

It is worth noting that our work focuses on integer-like (or handle-like) explicit dependencies due to their prevalent occurrence and the fact that Syzkaller only supports handlers of the integer type for explicit dependencies. By recognizing the insertion and lookup operations that operate on the same global data container, SyzGen++ effectively recovers explicit dependencies (see §3 for more details). To this end, SyzGen++ performs symbolic execution on each syscall interface separately to record symbolic access paths, which are derived from memory reads on writes and formalized in §3.2.3. Subsequently, SyzGen++ identifies any pairs of insertion and lookup operations against the same data container by matching their corresponding symbolic access paths. To reduce false positives, we propose a novel lightweight trial-and-error approach based on the observation that sequential allocation is commonly used for the "handler" (*e.g.,* file descriptor). Additionally, we develop a policy that selectively concretizes (or symbolizes) memory to balance exploration and scalability.

We evaluated SyzGen++ against Linux and macOS drivers and identified many new dependencies that were missed by either previous solutions or manually-curated syscall specifications. We show that they result in significantly higher coverage and more crashes.

In summary, we made the following contributions:

```
resource fd_rdma_cm[fd]
resource rdma_cm_id[int32]                    <- Explicit Dependencies
openat$rdma_cm(fd const[AT_FDCWD], file ptr["/dev/infiniband/rdma_cm"],
  flags const[O_RDWR], mode const[0]) fd_rdma_cm

rdma_create_id {
  cmd const[0, int32]                         <- Command Identifiers
  id  rdma_cm_id (out)
}
rdma_destroy_id {
  cmd       const[1, int32]                   <- Command Identifiers
  padding   const[0, int32]
  response ptr64[int32]
  id        rdma_cm_id (in)
}
write$CREATE_ID(fd fd_rdma_cm, data ptr[rdma_create_id],
  len bytesize[data])
write$DESTROY_ID(fd fd_rdma_cm, data ptr[rdma_destroy_id],
  len bytesize[data])
```

Figure 1: Excerpt from Syzkaller to describe syscall specifications in which we use 'resource' to declare an explicit dependency and 'in' and 'out' to specify the direction of a buffer or field.

- We propose a novel approach for the identification of explicit dependencies. Specifically, we define the two fundamental building blocks of *insertion* and *lookup* operations and their pairing.
- We develop a suite of techniques to realize the framework. This includes leveraging symbolic access paths, a lightweight trial-and-error-based method to generate specifications of dependencies, and a selective symbolic execution to balance exploration and scalability.
- We implemented a prototype, dubbed SyzGen++, that supports both macOS and Linux. We open source our tool at https://github.com/seclab-ucr/SyzGenPlusPlus.git to support future research in this area.
- We demonstrate through empirical results that SyzGen++ found 245 more dependencies for drivers with no traces. We compare SyzGen++'s performance to that of DIFUZE, KSG, and SyzDescribe in terms of code coverage, achieving an average improvement of 71%, 67%, and 39%, respectively. Additionally, We found 10 previously unknown bugs in Linux 6.2 and got 6 CVEs, using specifications generated by SyzGen++.

2. Background and Related Work

2.1. Device Drivers

To support communication between userspace and drivers provided by different vendors, modern OSes specify a few generic syscalls such as `write` and `ioctl` (or its counterpart `IOConnectCallMethod` for macOS IOKit drivers [16]) that can receive arbitrary driver-specified structures as input. For instance, the prototype of `ioctl` is defined as follows:

```
int ioctl(int fd, unsigned long request, void* arg
    );
```

Tool	Target	Technique	Requirement			Dependency Inference	Type Recovery	Constraint Recovery
			Source Code	Trace	Spec			
DIFUZE [7]	Linux Driver	Static Analysis	✓	✗	✗	✗	✓	✗
HFL [13]	Linux Driver	Concolic Execution	✓	✗	✗	✓$^{-}$	✓	✓
Moonshine [6]	Linux	Trace/Static Analysis	✓	✓	✓	✓	✗	✗
KSG [8]	Linux Driver	Clang Static Analysis	✓	✗	✗	✗	✓	✓
FUZZNG [14]	Linux Driver	API Hooking	✓	✗	✗	✓*	✗	✗
SyzDescribe [15]	Linux Driver	Static Analysis	✓	✗	✗	✓*	✓	✗
IMF [6]	macOS	Trace Analysis	✗	✓	✓	✓	✗	✗
SyzGen [9]	macOS Driver	Symbolic Execution	✗	✓	✗	✓	✓	✓
SyzGen++	Linux/macOS Driver	Symbolic Execution	✗	✗	✗	✓	✓	✓

$^{-}$: HFL's heuristic only works for 5 out of 16 drivers in our evaluated dataset.
*: FUZZNG and SyzDescribe focus on only a subset of explicit dependencies that involve file descriptors.

TABLE 1: Comparison of recent fuzzing techniques on interface recovery. SyzGen++ is the only tool capable of inferring dependencies, recovering structures, and identifying constraints, all without relying on source code or pre-existing traces

where the first argument `fd` is a file descriptor for a specific device obtained by calling `open` with its device file name, the second argument is an integer commonly known as the *command identifier*, and the type of the third argument varies significantly depending on the implementations of drivers and the command identifier. One driver can provide multiple functionalities through this unified entry point, and hence determine which one is desired based on the command identifier. We refer to each functionality bound to a specific command identifier value as a separate **interface**. The separation of interfaces within a driver is consistent with the convention of Syzkaller's specifications, allowing us to specify different types for different functionalities and explicit dependencies between them. As depicted in Fig. 1, we append different suffixes (*e.g.,* `CREATE_ID` and `DESTROY_ID`) to the same syscall to indicate different interfaces. It is worth noting that the command identifier does not necessarily need to be passed as an individual argument and can be embedded into a larger complex structure.

2.2. Kernel Fuzzing

Fuzz testing is an automated technique for discovering vulnerabilities that randomly generates test inputs and feeds them to the target program until it crashes. For syscalls that require complex nested structures as inputs and heavily sanitize the user-provided data for security concerns, a naive fuzzer is unlikely to produce valid inputs that could reach deep code, resulting in low code coverage. To address this, Syzkaller [5] allows users to develop syscall specifications in Syzlang, a strongly typed language to specify the structures and constraints of the inputs and the relationship between fields (*e.g.,* one field indicates the length of another one), as shown in Fig. 1. Specifically, the resource type in Syzlang represents a "handler" (*i.e.,* explicit dependency) produced by the kernel. Other types (*e.g.,* `const`) are self-explanatory. Implementing specifications is a manual and time-consuming process, particularly when the source code is unavailable.

Table 1 provides an overview of state-of-the-art tools for automating the process of interface recovery. In the rest of this section, we provide a detailed discussion of this summary.

Type and Constraint Recovery. In the realm of syscall specification, the most fundamental aspect is to define the types and constraints of syscall arguments to guide fuzzers in generating mostly valid inputs. DIFUZE [7] performs static analysis on source code, specifically LLVM IR, to extract accurate input types, while NTFuzz [17] conducts static analysis on documented Windows API functions that call undocumented syscalls to understand how inputs are constructed. A wealth of literature exists on reverse engineering of variable types for binary-only programs [18]–[20], which can be applied to proprietary drivers. In terms of recovering constraints on user inputs, prior work [8], [9], [13] leverages symbolic execution. As an alternative to type recovery, CoLaFuzz [21], V-Shuttle [22], and FUZZNG [14] propose to hook all `copy_from_user`-like functions to directly inject data instead of generating inputs with complex nested structures, effectively decoupling a multi-layer pointer into a sequence of one-layer buffers.

Explicit Dependency Inference. One of the factors that contribute to Syzkaller's success is its support for declaring explicit dependencies (or value dependencies) between syscalls. For example, a valid invocation of the `read` syscall requires a preceding invocation of `open` and passing the returned file descriptor `fd` to `read` as the first parameter. Type matching [23], [24] is a trivial solution to infer explicit dependency when type information is available, but it only works for non-primitive types. IMF [6] and Moonshine [12] leverage existing traces collected from third-party applications or test suites to infer explicit dependencies by detecting identical values from the input and output of syscalls that are consistent across traces, *e.g.,* the value returned from `open` always matches with the first parameter of `read` called right after it. FuzzGen [25] and WINNIE [26] perform static analysis on consumer programs to construct data dependency graphs and recover the dependencies between API calls. However, it is non-trivial to collect traces or consumer programs for kernel drivers, as reported in SyzGen [9]. Instead, SyzGen proposes first to recover some dependencies from a small set of traces and then discover more dependencies for interfaces without any traces, as interfaces within the same driver usually share common code or have similar logic on how to process dependencies.

One notable exception that does not require any existing

traces is HFL [13], which is a hybrid Linux kernel fuzzer that leverages concolic execution to monitor every comparison instruction, where one operand comes from userspace and the other one is some global data previously copied back to userspace. The underlying assumption is that the dynamically allocated handler must be stored in a global variable, copied back to userspace, and then used to check against the user-provided handler. This approach is not practical because it requires driving the execution precisely to explore certain paths of syscalls with valid inputs and its assumption only holds in 5 out of 16 drivers in our evaluation (see §5.2 for more discussion). Another exception is FUZZNG [14] and SyzDescribe [15], designed to identify special dependencies like file descriptors and sockets by recognizing kernel APIs responsible for handling them (*e.g.,* `fdget`). The difference between the two is that FUZZNG uses dynamic analysis (*i.e.,* API hooking) while SyzDescribe only employs static analysis. `SyzGen++` does not need to rely on any domain-specific knowledge and can identify a wider range of dependencies.

Implicit Dependency Inference. Unlike explicit dependency, implicit dependency (or ordering dependency) is more subtle and mandates a certain sequence of syscalls but does not involve any `fd`-like handlers. IMF [6] and Moonshine [12] propose to preserve the order of syscall sequences from existing test suites. As mentioned in §1, Syzkaller constructs a priority table for all pairs of syscalls based on their frequencies of occurrence in the corpus. Healer [27] further refines this approach by proactively removing syscalls in a single testcase to observe the coverage changes and detect influential syscalls. Meanwhile, Moonshine performs static analysis to identify potential implicit dependencies by finding pairs of read and write operations on the same global variable. Similarly, Minerva, specific to browser API fuzzing, learns the implicit dependencies of APIs by leveraging dynamic mod-ref analysis. Additionally, ACTOR [28] captures actions that potentially operate on shared data structures across different syscalls and then synthesizes programs based on manually-written templates in a domain-specific language that express the implicit relationship among actions. In this work, we focus solely on explicit dependencies and aim to automatically generate syscall specifications in Syzlang, which can be directly used by tools built atop Syzkaller, such as Healer, Moonshine, and ACTOR.

3. Overview

In this section, we present two motivating examples that have been simplified from real-world cases. These examples aim to illustrate our key observations regarding how the kernel typically implements explicit dependencies. Subsequently, we provide a detailed description of the workflow of `SyzGen++`, along with the intuitions behind our approach to infer explicit dependencies.

```
1  def create_request(gService):
2      int idx = gService->nextID++;
3      void* request = alloc_request();
4      gService->requests[idx] = request; # insertion
5      return idx;

6  def get_request(idx, gService):
7      return gService->requests[idx]; # lookup
```

(a) Function `create_request` creates a kernel object, stores it in the next available slot in an array, and returns its index as the 'handler', while `get_request` can retrieve the corresponding object by looking up the array using user-provided 'handler'.

```
1  def create_link(gService):
2      int idx = gService->nextID++;
3      Node* node = alloc_node();
4      node->id = idx;
5      last = gService->head;
6      for (Node* e = last->next; e; e = e->next)
7          last = e;
8      last->next = node; # insertion
9      return idx;

10 def get_link(idx, gService):
11     for (Node* e = gService->head->next; e; e = e->next)
12         if (e->id == idx) # lookup
13             return e;
14     return NULL;
```

(b) Function `create_link` creates a kernel object, appends it to the end of a linked list, and assigns a sequentially allocated ID as the 'handler', while `get_link` iterates through the list to find the corresponding object that has the same ID as the user-provided one.

Figure 2: Two motivating examples of explicit dependencies from macOS IOKit drivers (IOBluetoothHCIUserClient and AudioAUUCDriver). Both of them involve insertion and lookup operations.

3.1. Motivating Examples

Fig. 2a illustrates the interface `create_request` that allocates a new object and inserts it into a global array of which the index is returned to user space as a "handler" to avoid exposing internal kernel data. Subsequent syscalls that need to operate on the same object can retrieve it from the global array based on the previously returned "handler". Similarly, Fig. 2b shows the interface `create_link` that maintains a global linked list, where each newly created node is appended to the end. It also assigns a unique ID to each element so that the other interface `get_link` can retrieve the corresponding object by examining each element in the list. Despite their different implementations, we made the following three key observations from the common pattern:

- In user space, an explicit dependency only involves an integer (or "handler") which corresponds to a complex object maintained in kernel space.
- In kernel space, an explicit dependency must involve a

producer that creates a complex object and stores it in some global data container and a consumer that retrieves the corresponding internal object by looking up the global data container based on the user input.

- In kernel space, the producer must associate each object with a unique "handler" and pass it back to user space, enabling subsequent syscall invocations to indicate which object to use.

Intuitively, a kernel object is a data structure that represents a system resource that cannot be directly accessed by user programs, *e.g.,* a file, thread, or socket. Instead, user programs need to request kernel intervention to examine or modify the resources on their behalf. To identify which system resource to access, applications must obtain a unique identifier, also known as a 'handler', that corresponds to an internally-maintained data container. This identifier could be the index of an object in an array as demonstrated in Fig. 2a, or a unique ID that is assigned sequentially and bound to an entry in a linked list as shown in Fig. 2b. Moreover, different types of resources are typically maintained in different data containers. Based on our observations, we can abstract two essential operations for any explicit dependency: *insertion and lookup upon the same data container*. It is straightforward to determine whether these operations operate on the same data container by examining the symbol names and how the kernel accesses it (if source code is available). In the first motivating example, both the producer and consumer access the arrays named `requests` and derived from the same global variable `gService`. The same procedure can also be applied to binaries.

3.2. System Architecture

Fig. 3 illustrates the workflow of `SyzGen++`, which aims to automatically infer explicit dependencies between driver interfaces and generate specifications for model-based fuzzers. At a high level, `SyzGen++` first identifies all target drivers and their entry functions (§3.2.1). Then, for each driver, it performs symbolic execution to recover the type and constraints of the syscall inputs (§3.2.2). In the meantime, it collects all memory operations to recognize insertion and lookup operations via pattern matching and then identifies the pairing of them as dependency candidates if they operate on the same data container (§3.2.4). The pairs are then validated to infer and generate dependencies (§3.2.4), and the tool generates specifications for fuzzing based on the results. In the rest of this section, we discuss each component in detail.

3.2.1. Device and Entry Identification. The initial step of `SyzGen++` involves detecting potential driver targets that can be fuzzed on physical or virtual machines. Subsequently, `SyzGen++` employs established techniques from prior research [9], [21] to extract the entry functions for further program analysis. For macOS IOKit drivers, `SyzGen++` utilizes the `IOServiceMatching` API to query the OS and discover all available drivers. Further, it extracts

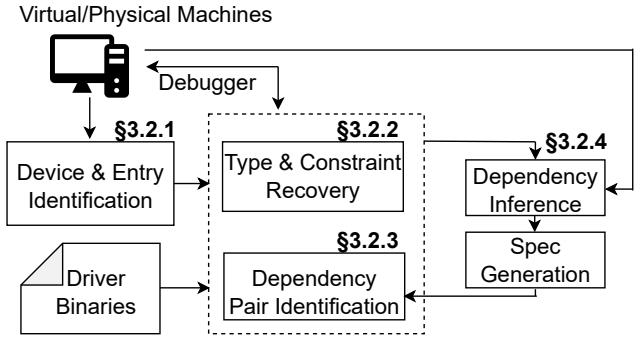

Figure 3: The workflow of `SyzGen++`: first, it discovers drivers running on the device and their entry points (§3.2.1). Next, it performs symbolic execution to recover the types and constraints of the syscall inputs (§3.2.2), and to identify dependency pairs (§3.2.3). The pairs are then validated to infer and generate dependencies (§3.2.4), and the tool generates specifications for fuzzing based on the results. It is possible to refine the specifications iteratively.

their entry functions by searching for specific functions (*e.g.,* `ExternalMethod`) in binaries based on the official IOKit design guidelines [29]. In contrast, for Linux drivers, `SyzGen++` recursively scans the device folder (*e.g.,*/dev) to obtain all available device files that can be successfully opened to acquire file descriptors. These file descriptors, in turn, can then be utilized to obtain the corresponding `struct file_operations` containing all the entry functions.

3.2.2. Type and Constraint Recovery. `SyzGen++` follows conventional driver development practices, such as using the second argument of `ioctl` as the command identifier to extract all valid command values corresponding to each interface through symbolic execution. Additionally, to handle cases where the command identifier is embedded within other inputs, `SyzGen++` identifies the critical variable whose value comes directly from user space and influences the invocation of different functions [9]. `SyzGen++` then performs symbolic execution on each interface separately by imposing a constraint on the `command identifier` (*e.g.,* cmd == ONE_COMMAND_VALUE), during which it collects constraints on the user inputs and monitors every "use" of them (symbolized beforehand) to identify separate fields at byte granularity [9], [18]. To recover multi-layer pointers, `SyzGen++` intercepts all `copy_from_user`-like APIs, since all user data must go through these APIs to traverse the boundary between user and kernel space. Due to space constraints, we omit details on how to generate specifications from recovered types and constraints and direct interested readers to SyzGen [9].

3.2.3. Dependency Pair Identification. Based on the observations described in §3.1, we propose to detect explicit dependencies by recognizing two operations (*i.e.,* insertion and lookup) that meet the following three criteria: (1) the in-

sertion operation creates a new heap object and stores it; (2) the lookup operation retrieves different objects depending on the user input, and (3) both insertion and lookup operate on the same data container reachable from a global variable. We refer to the interface that performs the insertion operation as the producer and the one that performs the lookup operation as the consumer. Since each interface is analyzed separately, the third criterion is used to link the consumer and producer interfaces by identifying the shared data container they use.

For instance, in the first motivating example illustrated in Fig. 2a, `create_request` inserts a newly-created heap object into an array at line 4, while `get_request` obtains a specific one by looking up the array using user-provided handler (*i.e.,* an index to the array) at line 7. Both of them manipulate the same array (*i.e.,* `gService->requests`) reachable from a global variable. Similarly, the second example shown in Fig. 2b follows the same pattern except that its lookup operation requires a loop with a conditional check against each element in the linked list (from lines 11 to 13). Hence, `SyzGen++` detects the specified insertion and lookup as follows:

Identifying Insertion: `SyzGen++` records every executed memory write whose source is a pointer to a heap object that has been allocated within the same interface. All such memory writes are considered "insertion candidates". However, for a memory write to be considered a true insertion, it must be paired with a lookup (which will be described in the later matching phase).

Identifying Lookup: A lookup operation is defined as a procedure that attempts to find one specific object from a collection based on user input, which can be categorized into the following two types:

(1) Single-memory-read lookup: any single memory read that can potentially access different objects depending on the user input is considered a lookup, *e.g.,* `gService->requests[idx]`. To identify such lookups, `SyzGen++` conducts symbolic execution with all user inputs symbolized and extracts symbolic expressions for memory read addresses that are computed based on the user input, *e.g.,* `gService->requests + idx * sizeof(request)` is the symbolic expression for the memory read address at line 7 in Fig. 2a, in which `idx` is directly from the user input, resulting in accessing different objects when `idx` varies.

(2) Multiple-memory-read lookup: it iterates through the container and checks for the presence of a desired element, typically by comparing a specific field of the element against the 'handler' provided by the user, *e.g.,* lines 11–13 in Fig. 2b. `SyzGen++` detects the existence of such a lookup based on the following criteria: it must execute multiple comparison instructions that compare the same user input against the same field derived from different objects. Intuitively, each element within the container should have the same type and maintain its own ID (or 'handler') in a dedicated field, enabling the lookup operation to differentiate between them based on the field. Specifically, `SyzGen++` again conducts symbolic execution to collect symbolic constraints for all comparison instructions. One of the operands of these instructions must be directly derived from user input (e.g., `idx`), while the other is derived from an object. If `SyzGen++` identifies multiple objects with the same field compared against the same user input, it considers the corresponding comparison instructions to be part of a lookup operation.

It is worth noting that our algorithm is agnostic to the implementation of the underlying data container as it captures the high-level semantics of two fundamental operations – insertion and lookup. The only assumption in the multiple-memory-read lookup scenario is that each element should maintain its own 'handler', which assists `SyzGen++` in associating the lookup operation with user input.

Matching Insertion and Lookup: The objective of this step is to match the insertion and lookup performed on the same data container. Therefore, the initial step is to determine the same objects across different interfaces. To this end, we adapt the well-known technique, namely Access Paths, from programming languages [30]. An access path is a base variable followed by a finite sequence of field accesses, *e.g.,* `gService->head->next` and `gService->head->next->next` are two different access paths accessing different objects. By tracing back each access to its root (*e.g.,* some global variable), we can uniquely identify an object. In the absence of source code, we use the unique addresses of global variables and field offsets to represent an access path. For simplicity, we use symbols instead of offsets in the rest of the paper. In contrast to traditional access paths composed of variables and fields, `SyzGen++` also retains symbolic expressions in the access path such that one could refer to different objects, *e.g.,* `gService->requests + idx * sizeof(request)` where `idx` is user-controllable.

To simplify the description, we state the following definitions: a collection of objects can be modeled by a directed graph $G = (N, E)$. Each node $n_i \in N$ corresponds to an object, and an edge in G is a tuple $\langle n_i, f, n_j \rangle \in E$ connecting node n_i and n_j. Here, f is a symbolic function such that $f(n_i) = n_j$ and $f \in F$, in which F denotes all symbolic functions comprised of exactly one pointer dereference, any number of symbolic variables, and arithmetic operations, *e.g.,* $f(ptr) = *(ptr + idx * 8)$. It indicates that an object represented by n_j can be retrieved via the object represented by n_i. The symbolic function f is not unique since it is possible that f_i and f_j are semantically the same but syntactically different. For instance, $ptr + idx * 8$ is semantically equivalent to $ptr + (idx << 3)$. Note that a node can have more than one outgoing edge with a given f as it may involve symbolic variables resulting in different outcomes. Additionally, since there might be multiple ways to access a node n_k (*i.e.,* $\exists n_i, n_j \in N, f_i, f_j \in F, f_i(n_i) = f_j(n_j) = n_k$), we define a partial order between them as follows: $f_i(x_i) \preceq f_j(x_j)$ if $\langle n_i, f_i, n_k \rangle$ happens before $\langle n_j, f_j, n_k \rangle$ during a single execution path. This partial order enables us to define a partial inverse of f as f^{-1} such that $f^{-1}(n_k) = n_j$ if

$\exists f_j \in F, f_j(n_j) = n_k$ and $\forall f_i, n_i \in \{(f,n)|f \in F \wedge n \in N \wedge f(n) = n_k\}, f_i(n_i) \preceq f_j(n_j)$. Intuitively, when we backtrack a field to its root on a single execution path, we always reverse the latest dereference of the field to find its parent object.

A `symbolic access path` in G is a sequence of connected edges denoted as $f_{1...l}(n_1)$, which equates to $(\langle n_1, f_1, n_2\rangle, ..., \langle n_l, f_l, n_{l+1}\rangle)$. Node n_1 and n_{l+1} represent the source and destination of the path, respectively, and the source must be a global object that can be uniquely labeled. Two `symbolic access paths` are equivalent if they could access the same objects in the same order: $f_{1...i}(x_1) = f'_{1...j}(y_1) \Leftrightarrow x_1 = y_1 \wedge i = j \wedge \forall k \leq i, f_k(x_k) = f'_k(y_k)$.

As aforementioned, a symbolic function f may involve symbolized user inputs resulting in different outcomes and may be syntactically different. Thus, we use SMT (satisfiability modulo theories) solver to prove $f_k(x_k)$ and $f'_k(y_k)$ can access the same object, *i.e.,* $f_k(x_k) == f'_k(y_k)$. Note that we evaluate them in a top-down manner, meaning we evaluate $f_k(x_k) == f'_k(y_k)$ before $f_{k+1}(x_{k+1}) == f'_{k+1}(y_{k+1})$, and thus we use the same object x_{k+1} and evaluate $f_{k+1}(x_{k+1}) == f'_{k+1}(x_{k+1})$ instead.

Now that we know how to identify objects based on their symbolic access paths, let us revisit how we record insertion and lookup. For any memory write whose source is a pointer to a newly-created heap object, we backtrack the destination address to obtain its symbolic access path. Similarly, for any memory read, we backtrack the source address if it could point to different objects, *i.e.,* the symbolic access path $f_{1...i}(x_1)$ has different destinations. We identify a match between insertion $f^{write}_{1..i}(x_1)$ and lookup $f^{read}_{1...j}(y_1)$ if $f^{write}_{1...i}(x_1) = f^{read}_{1...j}(y_1)$. To reduce false positives (*i.e.,* mistakenly identifying lookup operations), we require that the last symbolic function f_i involves only one symbolized variable from user inputs and previous symbolic functions do not contain any user inputs. The rationale is that we observed there is always a deterministic access path for the underlying data container, and only one handler (usually a 32-bit integer from user inputs) is involved in determining the final object to access. In the first motivating example shown in Fig. 2a, we can collect a symbolic access path ($f^{write}(gService)$ = `gService->requests + 0`) for the insertion at line 4 when `gService->nextID` is initialized to zero, and a symbolic access path ($f^{read}(gService)$ = `gService->requests + idx * sizeof(request)`) for the lookup at line 7, in which only the last field access involves the user input following our observations. We could successfully match them as both access the same objects (*i.e.,* `gService` and `requests`) and $f^{write}(gService) = f^{read}(gService)$ when `idx` equals to zero.

As aforementioned, for the second type of lookup operation involving multiple instructions, `SyzGen++` focuses on comparison instructions and identifies potential lookup based on our observations. For each comparison instruction that satisfies our criteria (*i.e.,* one operand is directly from user input and the other one is a field derived from an object), we collect a tuple $\langle f^{read}_{1...i}, idx\rangle$ in which `idx` is a symbolic variable from user inputs and $f^{read}_{1...i}$ signifies the symbolic access path of the other operand of the comparison instruction. We then group all tuples based on the symbolic variables (*i.e.,* `idx`) and last field access (*i.e.,* f_i) to ensure that those from the same group indeed check the same field against the same user input. We denote each group as $S_{idx,f_i} = \{f_{1...j}|\texttt{there exists}\langle f_{1...j}, idx\rangle \wedge f_j = f_i\}$, and consider it as a lookup operation if the size of S_{idx,f_i} is larger than a threshold (*i.e.,* three in our experiments). Intuitively, the condition $f_j = f_i$ filters out those accessing different fields, and the unique `idx` helps prune irrelevant symbolic access paths. We say we find a match between the insertion and lookup if $\forall f_{1...j} \in S_{idx,f_i}$, $f_{1..j-1}$ is also a symbolic access path for the insertion. For example, in the second motivating example in Fig. 2b, we could collect a set of symbolic access paths for the insertion at line 8 when exploring different paths during symbolic execution: S_{write} = {`gService->head->next`, ..., `gSerivce->head->next-> ... ->next`}. Similarly, we collect multiple tuples for the comparison at line 12 and group them into a single set S_{idx,f_i} = {`gService->head->next->id`, ..., `gService->head->next->...->next->id`}. As we can see, every access path $f_{1...j}$ in S_{idx,f_i} has a corresponding access path $f_{1...j-1}$ in S_{write}.

3.2.4. Dependency Inference and Specification Generation. Upon identifying a pair of one producer with insertion and a consumer with lookup, we need to generate the corresponding specification that specifies the location of the handler in the input and output buffer (*e.g.,* offsets), as illustrated in Fig. 1. Determining the location of the handler in the input buffer is straightforward since the symbolic access paths always include a symbolic variable from user space (with corresponding offsets, *e.g.,* input[0] or input[4]), which informs us of the exact location of the handler. In contrast, symbolic access paths for insertion do not encode any user space inputs since the handler is generated by the kernel in the producer interface (*e.g.,* line 2 in Fig. 2a). Therefore, the location of the handler in the output buffer is unknown, and identifying it can be challenging since it is typically just an integer. Tracking all potential handlers from their creation sites to user space is impractical due to the vast number of candidates.

Instead, we propose a novel, lightweight trial-and-error approach that enumerates all possible offsets and verifies each hypothesis based on the observation that sequential allocation is often used for the handler, *e.g.,* file descriptors in Linux. Thus, `SyzGen++` leverages fuzzing to produce a test case that could successfully invoke the producer with no error code returned, indicating that it has produced a handler. `SyzGen++` discards this potential dependency if fuzzing fails within a pre-configured time budget (we empirically set a 30-minute timeout in our evaluation). Subsequently, `SyzGen++` proposes a hypothesis on the offset of the handler in the output buffer and verifies it based on the aforementioned method. It repeats the test case multiple

times to find sequential numbers in the output buffers at the hypothesized offset. If it fails, SyzGen++ repeats the process with every feasible offset until one test supports the hypothesis. If none of the hypotheses passes the validation, we discard this potential dependency and deem it as a false positive from the previous step.

It is worth noting that this approach works well in practice, as demonstrated in our evaluation (see §5.2). Furthermore, SyzGen++ flags special dependencies (*i.e.,* file descriptors) by monitoring certain APIs (*e.g.,* alloc_fd). These dynamically-created file descriptors can be used to obtain their entry functions by checking their associated `struct file_operations`, enabling SyzGen++ to generate dependency directly without needing to match a lookup operation. For instance, if the ioctl entry function is valid, SyzGen++ can directly generate a dependency between the interface that returns a newly-created file descriptor and the corresponding syscall (*i.e.,* `ioctl`), knowing that this file descriptor is passed as the first argument to ioctl.

4. Implementation

We have implemented SyzGen++ atop Angr [31] and SyzGen [9] with 12,587 lines of Python code for interface recovery and dependency inference, and ∼1K lines of Go code into Syzkaller for fuzzing and test case generation. It supports analyzing macOS IOKit drivers and Linux drivers without any existing traces or source code. Specifically, SyzGen++ employs symbolic execution to perform interprocedural and path-sensitive analysis on drivers' entry points, during which it collects all the accesses to user inputs to recover types, constraints on user inputs to refine the valid input ranges, and insertion/lookup operations for dependency identification. The majority of the code is for type and constraint recovery and taming symbolic execution as we will explain in the rest of the section. We omitted some details in the paper due to space limits and referred to SyzGen [9]. Additionally, we set a ten-minute timeout for each run of symbolic execution to avoid indefinite execution and model some common functions (*e.g.,* malloc and `copy_from_user`) from the core kernel to make it more performant and support type recovery. In total, we have modeled 256 functions for Linux and macOS, among which 175 (*e.g.,* `printk`) can be simply replaced with a dummy function. Furthermore, SyzGen++ does not target any specific kernel version. This is because it recovers the structure of input during symbolic execution, making it structure-aware and eliminating the need to know the driver structure in advance.

It is possible to have multiple dependencies within the same structure. For instance, three interfaces A, B, and C need to be invoked sequentially as follows: A → handler 'a' → B → handler 'b' → C, in which B consumes handler 'a' and generates another one 'b', requiring an input of type '`struct foo { int dep1; int dep2;..}`'. SyzGen++ can analyze these three interfaces separately in any order. Upon analyzing A and B, SyzGen++ discerns the dependency between them. It then re-examines B, given that its input structures/constraints have been updated, and subsequently identifies the second dependency between B and C.

Since SyzGen++ aims to generate specifications for device drivers, it only supports the most common interfaces (*i.e.,* `open`, `ioctl`, `read`, and `write` for Linux drivers, and `IOServiceOpen` and `IOConnectCallMethod` for macOS IOKit drivers), leaving others (*e.g.,* `mmap`) to future work. We also set a soft limit (*i.e.,* 8 in our experiment) on the maximum number of loop iterations, meaning it yields the current path exceeding the limit instead of terminating it. Following SyzGen, SyzGen++ generates a test case that can reach the entry point of the target interface (`ioctl` handler), takes a snapshot through kernel debuggers at the entry point as the context, and then continue the execution with symbolic execution. As with other symbolic execution-based solutions, SyzGen++ is susceptible to the notorious path explosion problem. SyzGen partially addresses this issue by leveraging a concrete snapshot to concretize all memory, except for the user inputs. However, this approach limits the paths it can potentially explore, as we will discuss in detail in §4.3. To alleviate this issue, we have devised specific techniques tailored to kernel drivers, which are described below.

4.1. Exploration Strategy

One promising approach to cope with path explosion is to guide the exploration toward paths of interest. As described in §3, SyzGen++ conducts symbolic execution for three purposes, including type recovery, constraint recovery, and dependency inference. We observe that data copying from user space to kernel (*e.g.,* `copy_from_user`) and most sanity checks against user inputs often occur at the beginning of syscalls. Therefore, SyzGen++ initially applies Breadth First Search (BFS) to explore all feasible paths equally and then generates a specification for each path.

Unfortunately, some type information and dependency operations (*i.e.,* insertion and lookup) cannot be discovered unless the analysis reaches the end of the syscalls. Functions like `copy_to_user` are usually invoked just before the syscall ends and are crucial to identify output buffers. To address it, we propose refining each specification generated in the first round with another run of symbolic execution, using the Coverage-Optimized Search (COS) strategy [32] that prioritizes paths with uncovered code. Notably, the second run of symbolic execution imposes the same constraints on the user inputs reserved from the previous run and thus effectively continues the execution from where it stopped previously (though it still starts from the entry function).

Furthermore, we observe that most paths differ only on global variables and their generated specifications are the same. Consequently, the total number of specifications that SyzGen++ needs to refine is manageable. Although other exploration strategies exist, we find that BFS combined with COS suffices in most cases and leave further optimization to future work.

4.2. Path Pruning

Another technique we realized to alleviate the path explosion problem is to prune paths that lead to error handling as early as possible. SyzGen++ aims to generate specifications for valid inputs and thus invalid inputs failing to pass sanity checks are of less interest. Additionally, error handling code can be complex, resulting in substantial state forks and hence introducing significant overhead. We observe that modern OSes specify a set of special negative values as error codes, and it is common to pass an error code through the return value. Based on the insight, SyzGen++ examines the return value upon every return instruction and prunes paths that return error codes. It is possible that a function that does not return a value uses the return register (*e.g.,* eax for x86) for internal computation and happens to set a negative value to the return register. To prevent mistakenly pruning these paths, SyzGen++ also checks the immediate basic block at the call site to ensure the return value is indeed used. Note that those pre-defined error codes are different from valid pointers and can be differentiated easily (*e.g.,* Linux uses IS_ERR(void *ptr)).

Furthermore, SyzGen++ conducts a backward intra-procedure analysis to identify basic blocks that lead to returning an error code and thus could terminate paths even before they reach the return instruction. This additional analysis helps eliminate those paths as early as possible.

4.3. Selective Symbolic Execution

Symbolic execution requires making decisions about which data should be symbolized or concretized. This decision affects not only the scalability but also access path collection. Conventional under-constraint symbolic execution [33] can directly start from any function but require symbolizing all unknown data (*e.g.,* global variables), leading to exploring infeasible paths and exacerbating the path explosion problem. Moreover, it cannot resolve function pointers initialized inside other interfaces, which blocks deep code exploration. In contrast, S2E [34] proposes in-vivo symbolic execution, leveraging a full system emulator to execute the kernel concretely and only needs to symbolize user inputs. SyzGen [9] further augments it by generating proper test cases that guide the kernel to reach a certain state, *e.g.,* setting global variables properly. While concretizing the kernel state helps resolve pointers and reduce symbolized memory, it limits the paths that can be explored, *e.g.,* a global boolean variable can be toggled to trigger different behaviors via another interface, but we may fail to drive the kernel to reach both states. More importantly, it may prevent access path collection to identify lookup operations. For instance, in the second motivating example shown in Fig. 2b where it iterates through a linked list to obtain the target object, we cannot enter the loop if global data is concretized because the list is empty unless we know how to prime the test case to set up the kernel properly.

To balance this delicate relationship, we take advantage of both approaches and propose a hybrid solution that performs in-vivo symbolic execution but selectively symbolizes global data to unlock more paths to explore. Our current policy concretizes only two kinds of data: (1) global data that reside in certain areas (*e.g.,* const sections); (2) pointers that SyzGen++ cannot be resolved. This allows us to achieve reasonable scalability while still collecting access paths with relatively free exploration.

Note that even with our concretization strategy, reaching the end of a producer can still be difficult due to path explosion. This is why we had to resort to the dynamic trial-and-error approach to determine the offset of the handler in the output buffer, as described in §3.2.4.

5. Evaluation

In this section, we present our empirical test results to answer the following questions:

- How many dependencies SyzGen++ can infer compared to prior work (§5.2)?
- How accurate are the syscall specifications automatically generated by SyzGen++ (§5.3)?
- What is the performance of symbolic execution (§5.4)?
- Can SyzGen++ find more vulnerabilities (§5.5)?

5.1. Evaluation Setup

We consider the vanilla Syzkaller as the baseline and compare SyzGen++ against five other state-of-the-art kernel fuzzers, namely SyzGen [9], DIFUZE [7], KSG [8], Moonshine [12] and SyzDescribe [15]. We obtained the original traces from SyzGen and Moonshine, whereas we received the KSG executable from its authors for testing. Unfortunately, we did not receive any responses from the authors of FUZZNG [14], and as a result, we excluded it from the evaluation. Nevertheless, we believe that SyzGen++ has the potential to outperform FUZZNG, as it only handles a specific type of explicit dependencies (*i.e.,* file descriptor). For HFL, we attempted to feed it with the minimum specifications that contain driver interfaces without type and constraint information, but our preliminary result showed that it significantly underperformed compared to the vanilla Syzkaller. We contacted its authors to get their perspectives. It turns out that one static analysis component crucial to dependency inference and symbolic execution is missing from its repository. Thus, we excluded HFL from our evaluation.

We also found some deficiencies in DIFUZE when porting it to a newer version of LLVM and Linux kernel, making it broken for some drivers. Thus, we improved DIFUZE by fixing bugs and correcting some hard-coded domain knowledge to support newer kernel versions. Moreover, DIFUZE does not open source the component responsible for specification generation, and we also re-implemented it to have an end-to-end comparison. The enhanced version is referred to as DIFUZE^{+}.

Dataset. We evaluated the accuracy of explicit dependency inference on two datasets, one from SyzGen containing 25

Linux Driver	#Dependencies		
	Ground Truth^{+}	Syzkaller	SyzGen++
autofs	11	11(+3)	11
dri	23	**22**(+9)	7
fuse	17	17	1
kvm	94	**93**(+19)	94
loop_control	2	**1**	2
ppp	1	**0**	1
ptmx	108	108	106
rdma_cm	21	21	20
Total	277	304	242

$^{+}$: We collected the ground truth by cross-checking the source code, Syzkaller's specifications and SyzGen++'s results.

TABLE 2: Comparison of dependency inference between manually-crafted specifications and SyzGen++. In the Syzkaller column, the numbers within parentheses denote the false positives we manually identified given the default Linux configuration from syzbot. The preceding bold numbers that are smaller than the ground truth indicate missing dependencies that were uncovered by SyzGen++.

macOS Driver	Has Traces?	#Dependencies	
		SyzGen	SyzGen++
AudioAUUCDriver	Yes	5	5
AppleAPFSUserClient	Yes	21	21
AppleUpstreamUserClient	Yes	5	5
IOBluetoothHCIUserClient	Yes	235	170
IONetworkUserClient	Yes	5	5
AppleUSBHostFrameworkDevice	No	0	**1**
AppleUSBHostFrameworkInterface	No	0	**1**
AppleUSBHostInterface	No	0	**1**
Total		271	209

TABLE 3: Comparison of dependency inference between SyzGen and SyzGen++. We exclude targets where neither SyzGen nor SyzGen++ could find any dependencies. SyzGen++ has comparable performance when traces are available (except the IOBluetoothHCIUserClient case) and surpasses SyzGen in cases where traces are absent. Note that for tools such as SyzGen, Moonshine, and IMF that require traces, they are unable to identify any dependencies if no traces are provided.

macOS IOKit services and another one for Linux which includes eight Linux drivers with dependencies, as shown in Table 3 and Table 2, respectively. Note that most drivers are simple and only contain explicit dependencies that are trivial to learn, *e.g.,* the return value of 'open' is the first argument of 'ioctl'. This domain knowledge is encoded in most tools including DIFUZE, SyzDescribe, and our tool. We would like to evaluate all work against the difficult ones and thus constructed the second dataset by checking the existing specifications from Syzkaller to find drivers with non-trivial dependencies, assuming the handcrafted specifications were mostly correct. Additionally, since our work focuses on device drivers that lack support for specifications due to their large volume, our current prototype only supports analyzing `open`, `ioctl`, `read`, and `write` syscalls and their relevant dependencies. Consequently, we did not include other subsystems (*e.g.,* bpf and network) and collected eight drivers in total.

We ran all the following experiments on two machines, one Ubuntu 18.04 LTS equipped with Intel(R) Xeon(R) Gold 6248 CPU (having 80 2.50GHz cores) and 394 GB RAM, and one Macbook Air with 2.2 GHz Intel Core i7 and 8GB RAM. Our tool relies on SyzGen to support macOS, which currently only supports IOKit drivers and a VMware Fusion-based macOS setup. As VMware Fusion does not support macOS 12.3 Monterey or later at the time of our evaluation [35], we were limited to testing on the same older version (macOS 10.15.4 and VMware Fusion 11.5.7) as used by SyzGen. As for the Linux kernel, we chose Linux 5.15 with the same configuration from Syzbot and ran it inside QEMU. Each fuzzing campaign utilizes four CPU cores (*i.e.,* two QEMU instances with two CPU cores each) and tests one particular driver for 24 hours with five repetitions.

5.2. Explicit Dependency Inference

To assess the accuracy of explicit dependency inference, we first compared SyzGen++ against Moonshine and manually-curated specifications in terms of the numbers of identified dependencies counted based on the number of consumers (*i.e.,* the interfaces that consume a 'handler'). We also manually collected ground truth data for Linux by inspecting the source code of tested drivers. Unfortunately, upon examining the traces provided by Moonshine's authors, we discovered that only five drivers are being exercised, none of which correspond to our tested drivers with dependencies. Hence, it is infeasible for Moonshine or any other trace-based solutions to detect any dependencies. As Moonshine was published in 2018, it may not include some new test suites added since then. Therefore, we examined the latest test suites from the same sources as mentioned in its paper, including Linux Testing Project (commit 63e8c1e) [36], Linux Kernel selftests (v6.5.0-rc4) [37], and Open Posix Tests (v1.5.1) [38]. As expected, these test suites cover 21 device drivers in total, among which only two (*i.e.,* kvm and fuse) have explicit dependencies, meaning that at least 58.3% of the dependencies shown in Table 2 would be missed by trace-based solutions.

In contrast, SyzGen++ can infer 242 out of 277 ground truth dependencies and we found four dependencies missed by existing handcrafted specifications. We have reported our findings to Syzkaller, resulting in the resolution of one issue. Interestingly, we observed that Syzkaller specifies 31 more dependencies than the ground truth. Upon further investigation, we determined that these additional dependencies are either false positives or related to code that is not enabled by its default configuration, as explained in detail in Appendix A.1.

Although HFL is the closest work to SyzGen++, we found that it did not perform well due to one missing static analysis component from its repository, thus preventing it from being included in our comparison. In practice, HFL suffers from scalability challenges because it relies on a full-system emulation solution S2E, which limits its ability to support other platforms (*e.g.,* macOS and Android). Additionally, HFL requires drivers to function properly inside

S2E, and we found that it does not leverage KVM, which can lead to S2E getting stuck at booting time when too many drivers are enabled. Moreover, it requires precise execution to explore specific paths of syscalls since it employs concolic execution and relies on fuzzing to explore the codebase. Assuming HFL operates perfectly, it can only detect explicit dependencies in 5 out of the 16 drivers shown in Table3 and 2. This is because HFL's assumption regarding the usage of 'handler' (see §2 for details) holds true only for those 5 drivers in our dataset.

Furthermore, we evaluate `SyzGen++` against the macOS IOKit drivers tested by SyzGen and present the results in Table 3. `SyzGen++` identifies 209 dependencies compared to 271 by SyzGen. We manually reverse-engineered the binaries to confirm the validity of the results. Notably, `SyzGen++` was able to identify more dependencies than SyzGen in three cases where no traces were available, thereby demonstrating its effectiveness. `SyzGen++` also exhibited comparable performance to SyzGen when traces were available, except for the driver `IOBluetoothHCIUserClient`. Upon further investigation, we discovered that some dependencies were handled by the firmware, which is not analyzed by `SyzGen++`. Note that this is a limitation of our current implementation, not the design. We believe this issue could be addressed with additional engineering efforts.

False Negatives. We investigated the reasons why `SyzGen++` failed to identify the 35 dependencies (compared to ground truth). They correspond to 16, 16, 2, and 1 false negatives for `dri`, `fuse`, `ptmx`, and `rdma_cm`, respectively. One prominent reason is that certain code is not reachable due to unresolved indirect call. As mentioned in §4, `SyzGen++` generates a simple testcase and runs it concretely inside a virtual machine to reach the entry point of the target interface (*e.g.,* `ioctl`) where it takes a snapshot as the context and leverages it to resolve function pointers. However, some function pointers are only initialized when certain hardware is available and registered. Although our approach can be applied to physical machines to circumvent this problem, it requires significant engineering efforts, which we leave for future work. Additionally, our assumption that the producer must create a kernel object and return its associated 'handler' within one interface may not hold in some cases, where the interface that performs the object creation is different from the one that actually returns its 'handler'. For instance, in the dri module, `ioctl$DRM_IOCTL_GEM_FLINK` creates and returns a global ID for a specified object created in another interface `ioctl$DRM_IOCTL_GEM_OPEN`. In such cases, correlating these two interfaces is not supported yet. Furthermore, `SyzGen++` focuses on analyzing `open`, `ioctl`, `read` and `write`, and thus misses dependencies in drivers like `fuse` where additional syscalls (*e.g.,* `mount`) unlock more code.

False Positives. Our solution did not produce any false positives for the tested drivers, indicating the robustness of our approaches, including identifying insertion and lookup operations as dependencies (as discussed in §3.2.3) and verifying the sequential allocation of "handler" (as described in §3.2.4). Interestingly, we found only two drivers, namely `rdma_cm` and `kvm`, have false positives if we only rely on the pairing of insertion and lookup. We provide a detailed explanation in Appendix A.2.

Moreover, it is worth noting that all the dependencies identified by `SyzGen++` adhere to the sequential allocation pattern, and we have not encountered any counterexamples, except the one in `key` module demonstrated below and another one involving string type "handlers", which are outside the scope of our tool and its capabilities (see limitations in §6 for more details).

Generality of Our Approach. Our algorithm captures the high-level semantics of the two fundamental operations for explicit dependency, making it agnostic to the underlying data container. We manually examined the source code or disassembled binaries of the drivers for cases where `SyzGen++` successfully identified dependencies, and discovered six types of data containers: six drivers use one-dimension array, two use linked list, one uses XArray (eXtensible Arrays, ∼2K LOC in Linux), three use radix tree (which uses XArray internally and has ∼3K LOC), one uses OSArray and three use OSSet. The latter two are C++ classes for collections. Additionally, out of the 16 drivers with dependencies, 11 were detected based on the single-memory-read lookup that uses the index of an array (or any advanced data structures built on Array) as the 'handler'. For the rest 5 cases following the multiple-memory-read lookup, all of them maintain a 'handler' inside each object and use a counter as the 'handler', which is incremented by some constant every time a new object is allocated.

To further demonstrate our approach works for the most common and prevalent cases, we checked the existing specifications from Syzkaller for all other syscalls that `SyzGen++` does not support and collected 484 producers that could generate 'handlers' (*i.e.,* resources in Syzlang's terminology). We randomly sampled 50 cases and inspected their source code to verify whether they follow the patterns of insertion/lookup and sequential allocation. As a result, we found 37 out of 50 cases matched our proposed patterns. For the remaining unmatched ones, 9 requires correlating more than two syscalls as aforementioned, *e.g.,* `bpf$BPF_PROG_GET_NEXT_ID` returns the ID (or 'handler') of the next eBPF program previously loaded via `bpf$PROG_LOAD`, three do not meet our requirements for lookup, *e.g.,* `clock_getres` finds the resolution (precision) of the specified clock, which is not used in a lookup operation, and one does not follow the pattern of sequential allocation, *i.e.,* `add_key` adds a key to the kernel's key management facility and returns its serial number which is randomly assigned, and upon further investigation, we found that this is intentional to prevent covert channel problems, according to the comments in Linux.

5.3. Effectiveness of Interface Recovery

To evaluate the end-to-end performance of `SyzGen++`, we compared its fuzzing results with those of Syzkaller, enhanced DIFUZE$^+$, KSG, and SyzDescribe, as presented

Driver	#Avg. Coverage						p-value				
	Syzkaller	DIFUZE$^+$	KSG	Desc*	SyzGen++ w/o dep infer	SyzGen++	Syzkaller	DIFUZE$^+$	KSG	Desc*	SyzGen++ w/o dep infer
autofs	3427	N/A	2866	2325	3374	3651	0.004	N/A	0.004	0.004	0.004
dri	20260	2988	9575	13877	12107	17762	1	0.004	0.004	0.004	0.004
fuse	17867	2635	2355	2195	2566	3064	1	0.028	0.004	0.004	0.008
kvm	19498	11458	8761	14423	11611	16370	1	0.004	0.004	0.004	0.004
loop_ control	5690	9037	7554	7185	9179	10227	0.004	0.004	0.008	0.004	0.075
ppp	7556	7416	6527	7203	7446	7389	1	0.5	0.004	0.004	0.79
ptmx	16487	10754	14751	10596	21417	24620	0.004	0.004	0.006	0.004	0.004
rdma_cm	5840	2238	2678	N/A	7513	7880	0.004	0.004	0.004	N/A	0.004

Desc* stands for SyzDescribe.

TABLE 4: Comparison between SyzGen++ and previous work, encompassing Syzkaller, DIFUZE$^+$(*i.e.,* DIFUZE with bug fixes), KSG, and SyzDescribe. We conducted five 24-hour trials for each driver, and report the average code coverage. P-values were calculated using the Mann-Whitney U test on coverage. For the ablation study, we incorporated SyzGen++ w/o dependency inference.

Driver	#Avg. Crashes					
	Syzkaller	DIFUZE$^+$	KSG	SyzDescribe	SyzGen++ w/o dep infer	SyzGen++
autofs	0	N/A	0	0	0	0
dri	4	0	0	0	0.2	2.2
fuse	0	0	0	0	0	0
kvm	3.2	0.4	0	0.8	0.4	2
loop_ control	0	2	1.2	1	2.0	3
ppp	0.6	0	0	0	0.6	0.6
ptmx	4.0	3.4	0	0.4	0.4	1.6
rdma_cm	0.6	0	1.0	N/A	0.6	1.6

TABLE 5: Comparison between SyzGen++ and previous work in terms of the average number of unique crashes. For each driver, we conducted five 24-hour trials.

in Table 4 and Table 5. Although we fixed some prominent bugs in DIFUZE to support Linux 5.15, it still failed to recover device file names for four out of eight drivers due to its static analysis approach with ad-hoc domain knowledge. In contrast, our dynamic approach is more accurate. Nevertheless, DIFUZE$^+$successfully identified the ioctl entry points for seven out of eight drivers and produced the corresponding specifications. To give it a fair chance, we corrected the device file names for those four broken drivers, enabling us to measure the overall effectiveness of interface recovery by comparing the fuzzing results. We ran five trials for each target and applied the Mann-Whitney U test, as suggested by [39], to compare the performance.

As we can see from Table 4, for all drivers except ppp, SyzGen++'s improvement on coverage is statistically significant (*i.e.,* p-value < 0.05) compared to DIFUZE$^+$, achieving 71% coverage improvement on average. SyzGen++'s performance is comparable to DIFUZE$^+$on the ppp driver, likely because the driver has only one explicit dependency, which limits the potential contributions of dependency inference. Additionally, it is noted that ppp uses sequential allocation for the handler, starting from zero, and Syzkaller is optimized to favor special values including zero when randomly generating an integer. Similarly, SyzGen++ outperforms both KSG and SyzDescribe significantly in all tested drivers, resulting in 67% and 39% coverage improvement on average, respectively. This is expected since KSG does not consider dependencies, while SyzDescribe only deals with special dependencies like file descriptors and omits constraints on the inputs. As shown in Table 5, more code coverage typically leads to more crashes.

Surprisingly, SyzGen++ surpasses Syzkaller with handcrafted specifications in four of the eight cases tested. The main reason for this is that SyzGen++ identified more command values than Syzkaller, which unblocked more code. Take the module ptmx as an example, we found that some command values are hidden behind indirect function calls, presumably overlooked by humans. Conversely, SyzGen++ underperforms Syzkaller for fuse and dri as it misses some dependencies. Regarding KVM, Syzkaller implements some helper functions specific to KVM that enable proper setup of virtual CPUs into a state that is sufficiently interesting.

To investigate the extent to which dependency inference benefits fuzzing, we conducted an ablation study by disabling dependency inference in SyzGen++. This effectively ruled out the difference introduced by type and constraint recovery. As shown in Table 4, SyzGen++ outperformed its inferior version in all cases except ppp for the same reasons mentioned earlier. The coverage improvement ranges from 5% to 47%. In general, the more dependencies a driver has, the greater the benefits that SyzGen++ can bring. Interestingly, compared to DIFUZE$^+$and KSG which do not support dependency inference, SyzGen++ without dependency inference still achieved 54% and 38% more coverage, respectively, indicating that SyzGen++'s structure and constraint recovery is more accurate.

5.4. Performance

Although generating specifications is a one-time effort and the interfaces of kernel drivers do not change frequently, we measure its overhead and break it down into three parts to better understand its performance, including symbolic execution, specification generation, and dependency generation as shown in Table 6. We set a 10-minute timeout

Driver	SE	SG	DG including fuzzing	#CMD	Total/Avg
autofs	9.4h	7.6h	2.4h	14	19.5h/1.4h
dri	14.7h	1.1h	1.3h	108	17.1h/0.2h
fuse	0.3h	<0.1h	0.03h	1	0.5h/0.5h
kvm	39.1h	24.9h	18.4h	109	82.4h/0.75h
loop_control	1.5h	0.8h	0.04h	3	2.4h/0.8h
ppp	3.6h	0.4h	0.4h	23	4.7h/0.2h
ptmx	12.2h	4.2h	3.4h	58	20h/0.3h
rdma_cm	13.8h	5h	3.2h	23	22h/1h

TABLE 6: Time cost of SyzGen++. SE, SG, and DG represent the time cost of symbolic execution, specification generation, and dependency generation, respectively. The last column shows the total time cost and the average cost per CMD.

for each run of symbolic execution, though this may be exceeded because Angr (the symbolic execution engine we use) does not support precise timeout yet. As mentioned in §4.1, the number of rounds of symbolic execution required for each interface (*i.e.,* CMD) depends on the number of generated specifications, as SyzGen++ needs to refine each specification using the Coverage-Optimized Search (COS) strategy. As a result, the overhead of symbolic execution is proportional to the number of CMD values and the complexity of the interface. For instance, we found that the driver kvm is the most expensive to analyze due to the following reasons: 1) it has the most CMD values; 2) this module is more complex than others, resulting in more constraints introduced on the inputs and hence more solving time in generating specifications; 3) due to missing bounds for nested array mentioned in §5.2, SyzGen++ initially identifies lots of false dependencies which are later pruned based on our heuristic of sequential allocation of "handler", leading to more fuzzing time in generating valid testcases for dependency validation. On average, for the 8 tested Linux drivers, each CMD takes 0.6 hours to finish and SyzGen++ generates 1.2 specifications per CMD.

We also conducted an evaluation of the effectiveness of the selective symbolic execution approach discussed in §4.3. The evaluation revealed that if we disable this optimization, SyzGen++ was unable to identify dependencies for four out of eight drivers, resulting in a 51% dependency miss rate. The primary reason for this was the inability of symbolic execution to resolve some function pointers, which impeded further execution. Additionally, path explosion was another common reason for missing dependencies. With the optimization enabled, we were able to eliminate some infeasible paths and alleviate this issue to some extent.

5.5. Bug Findings

To evaluate SyzGen++'s ability to uncover new bugs, we conducted a one-month fuzzing campaign on Linux kernel v6.2, using the same configuration as Syzkaller and specifications produced by SyzGen++. As mentioned in §5.1, SyzGen++ is built on top of SyzGen, which does not support the latest macOS version. We did not attempt to fuzz the old version (macOS 10.15.4) as it is less likely to yield new bugs, particularly considering the limited number of IOKit drivers available within a Virtual Machine that do not require root access. Furthermore, since Linux has been continuously fuzzed for years with Syzkaller, we do not expect to find many bugs in this code. Nonetheless, SyzGen++ found previously-unknown bugs in code fuzzed by Syzkaller. In addition, we found 10 other drivers, including bsg, btrfs, dlm, dm, dma, dmxdev, dvb, md, mtd, and ubi, which are enabled by Syzkaller's Linux config but do not have Syzlang specifications, and thus we ran SyzGen++ against them to generate corresponding specifications for fuzzing. In total, SyzGen++ found 10 previously-unknown bugs as listed in Table 7. We have responsibly disclosed these bugs to the respective maintainers, and so far, we have received 6 CVEs and patched 5 of them. Our investigation revealed that 6 of the bugs came from drivers without any specifications in Syzkaller, despite using the same Linux configuration and not enabling additional drivers for testing, indicating the lack of manual efforts in implementing specifications for all drivers since it is time-consuming and requires domain expertise. Given the substantial amount of drivers that were not enabled in our test and do not have any specifications, we believe that SyzGen++ could continue benefiting the Linux Kernel community.

SyzGen++ found 4 bugs from drivers that necessitate accurate dependency analysis as indicated in the third column of Table 7. The only one found within KVM requires dependencies to trigger —- dependencies that were overlooked by Syzkaller but identified by SyzGen++. This does not mean accurate dependency inference is unnecessary for the following reasons: 1) those drivers with non-trivial dependencies already have manually written specifications and were fuzzed for years. Thus it is expected that we didn't find many bugs in those codes given that SyzGen++ only identifies four more dependencies missed by Syzkaller; 2) we only reported previously unknown bugs. We have tried fuzzing some old kernel versions (e.g., 5.15) and found more N-day bugs that need accurate dependency inference, but only zero-day bugs are worth reporting. Thus, we only chose Linux v6.2 as our target for bug finding. We expect to find more bugs requiring accurate dependency inference in Android given that it lacks support from Syzkaller. Next, we discuss some case studies.

Use-After-Free in KVM: The root cause of this bug is that KVM installs file descriptors to enable userspace to read VM and vCPU statistics, but it does not acquire a reference to the VM to ensure the VM and its vCPUs remain active until the statistics file descriptors are closed. To find the bug, SyzGen++ had to use independent syscalls to create a VM, request a file descriptor bound to the newly-created VM specified by its handler obtained earlier, free the VM, and finally read from this file descriptor to trigger the UAF. Although Syzkaller already has a handcrafted and comprehensive specification for KVM (893 LOC in Syzlang), it misses the critical dependency between requesting a file descriptor for VM statistics and reading from that file descriptor, which was revealed by SyzGen++.

Crash Type	Affected function	Driver	Status
UAF*	kvm_stats_read	kvm+	Confirmed
UAF	do_raw_spin_lock	rdma+	Confirmed
UAF	__put_mtd_device	mtd	Confirmed
Divide-by-zero*	ubi_attach_mtd_dev	ubi	CVE-2023-31085
Null-ptr-def	vidtv_mux_stop_thread	demux	CVE-2023-31081
Null-ptr-def	hci_uart_tty_ioctl	ptmx+	CVE-2023-31083
Deadlock*	dm_get_inactive_table	dm	CVE-2023-2269
Deadlock*	cec_transmit_msg	cec	Confirmed
Deadlock*	dvb_frontend_get_event	dvb	CVE-2023-31084
Deadlock	gsmld_write	ptmx+	CVE-2023-31082

+: Has explicit dependencies
*: Patched bugs

TABLE 7: Previously-unknown bugs found by SyzGen++. UAF is short for Use-After-Free.

Null-Ptr-Deref at hci_uart_tty_ioctl: This bug is a race condition in which one ioctl call sets the flag and then initializes a data pointer, while another ioctl call checks the flag before dereferencing the pointer. Due to race conditions, the second call may access the uninitialized pointer even when the flag is set. Although the existing manually-curated specification does not miss any dependencies, we find that it omits two valid command values necessary to trigger the bug. This part of the code may have been overlooked because it is concealed by some indirect calls. SyzGen++ can successfully extract those command values, as it leverages a dynamic environment to resolve most indirect calls (see §4 for details).

6. Limitation and Future Work

Aside from the false negatives mentioned earlier, we point out some other limitations of our proposed approach as follows. SyzGen++ assumes that all objects within a single data container are of the same type. This can lead to less accurate specifications when dealing with data containers that contain objects of different types, *e.g.,* file descriptors and their associated `struct file`. While the kernel can distinguish between different file types using the subfield `f_ops`, SyzGen++ treats all file descriptors as the same, as they are maintained within a single data container via specific APIs (*e.g.,* `alloc_fd` and `fget`). This can lead to generated specifications that are too generic, as they assume that any file descriptor can be used interchangeably. However, we note that this limitation is not critical for fuzzing, as coverage feedback can guide the fuzzer to produce valid sequences from the generated candidates.

Although we have observed one case involving a macOS driver where a string is used as the key to perform lookup operations, Syzkaller supports only the 'handler' of type int (int8, int16, int32, and int64 to be more specific), and thus we consider it out of scope. Even if Syzkaller supports it in the future, we anticipate difficulty in identifying such cases if we directly apply our proposed approach. This particular macOS driver serves as a key-value storage using a hashmap, and therefore retrieves an object based on the hash value of the given string, rather than the string itself. In this case, SyzGen++ would not be able to associate the input string with the lookup operation. One approach to tackle this challenge is to model these data containers by hooking all their APIs (*e.g.,* HashMap.get(string)), as the computation of the hash value is typically implemented internally as libraries that are exposed for other drivers to use. Moreover, there is prior work on inferring function names based on code semantics using code embedding [40], which we could potentially leverage to identify standard data containers in an automated manner.

Additionally, we would like to point out some other directions worthy of exploration. In this work, we only focus on explicit dependencies as Syzlang does not support specifying implicit dependencies. There is also prior work on learning implicit relations between syscalls by leveraging existing traces/test suites [6], [9], [12] or coverage [27]. Additionally, Syzlang does not support a particular dependency in which users are responsible for specifying a unique value for a 'handler'. It would be interesting to extend Syzlang to support these dependencies. Although there are some common drivers between Android and Linux, we found that they do not necessarily share the same interfaces and Syzkaller only maintains one specification for one driver without separation for different downstream distributions. Similarly, driver code may evolve, requiring different specifications for different versions. Since it is time-consuming for manual development of syscall specifications, let alone long-term maintenance upon new code changes, we believe we could apply SyzGen++ to keep specifications updated and maintain different ones for different kernel versions and distributions.

7. Conclusion

This paper introduces SyzGen++, aiming to infer explicit dependencies between syscalls and automatically generate syscall specifications with driver binaries only. We abstract two key operations, namely insertion, and lookup, to identify dependency and leverage a lightweight trial-and-error-based approach to validate it. Our evaluation demonstrates that SyzGen++ achieves comparable performance in terms of dependency inference for drivers with good test suites and identifies 245 more dependencies for drivers without test suites. Additionally, SyzGen++ outperforms DIFUZE, KSG, and SyzDescribe in terms of code coverage, achieving 71%, 67%, and 39% improvement on average, respectively. Additionally, we found 10 previously-unknown bugs in Linux 6.2 and got 6 CVEs, using specifications generated by SyzGen++.

Acknowledgment

The authors would like to extend their sincere appreciation to the anonymous reviewers for their insightful and valuable feedback, which greatly contributed to the improvement of this paper. This work was supported in part by the National Science Foundation under Grant #2155213, #1953933, and #1652954.

References

[1] J. Stoep, "Android: protecting the kernel," *Linux Securit Summit (August 2016)*, 2016.

[2] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in os kernels," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 661–678.

[3] X. Bai, L. Xing, M. Zheng, and F. Qu, "iDEA: Static analysis on the security of apple kernel drivers," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1185–1202.

[4] "Syzbot," 2023. [Online]. Available: https://syzkaller.appspot.com/upstream

[5] D. Vyukov, "Syzkaller: an unsupervised, coverage-guided kernel fuzzer," 2019.

[6] H. Han and S. K. Cha, "IMF: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2345–2358.

[7] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[8] H. Sun, Y. Shen, J. Liu, Y. Xu, and Y. Jiang, "KSG: Augmenting kernel fuzzing with system call specification generation," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 351–366.

[9] W. Chen, Y. Wang, Z. Zhang, and Z. Qian, "Syzgen: Automated generation of syscall specification of closed-source macos drivers," in *ACM CCS*, 2021.

[10] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," ser. USENIX Security, 2018.

[11] Z. Zhang, H. Zhang, Z. Qian, and B. Lau, "An investigation of the android kernel patch ecosystem," in *30th USENIX Security Symposium*, 2021.

[12] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS fuzzer seed selection with trace distillation," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 729–743.

[13] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid fuzzing on the linux kernel." in *NDSS*, 2020.

[14] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions," 2023.

[15] Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, and A. A. Sani, "SyzDescribe: Principled, automated, static generation of syscall descriptions for kernel drivers," in *In Proceedings of IEEE Security and Privacy (Oakland)*, 2023.

[16] J. Levin, *Mac OS X and iOS internals: to the apple's core*. John Wiley & Sons, 2012.

[17] J. Choi, K. Kim, D. Lee, and S. K. Cha, "NtFuzz: Enabling Type-Aware kernel fuzzing on windows with static binary analysis," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 677–693.

[18] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 391–402.

[19] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*, 2010, pp. 1–1.

[20] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," 2011.

[21] T. Mu, H. Zhang, J. Wang, and H. Li, "CoLaFUZE: Coverage-guided and layout-aware fuzzing for android drivers," *IEICE TRANSACTIONS on Information and Systems*, vol. 104, no. 11, pp. 1902–1912, 2021.

[22] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu, "V-Shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2197–2213.

[23] J. Jiang, H. Xu, and Y. Zhou, "RULF: Rust library fuzzing via API dependency graph traversal," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 581–592.

[24] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "FANS: Fuzzing android native system services via automated interface analysis," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 307–323.

[25] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2271–2287.

[26] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "WINNIE: Fuzzing windows applications with harness synthesis and fast cloning," in *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[27] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, "HEALER: Relation learning guided kernel fuzzing," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 344–358.

[28] M. Fleischer, D. Das, P. Bose, W. Bai, K. Lu, M. Payer, C. Kruegel, and G. Vigna, "ACTOR: Action-guided kernel fuzzing," 2023.

[29] "IOKit device driver design guidelines," 2023. [Online]. Available: https://developer.apple.com/library/archive/documentation/DeviceDrivers/Conceptual/WritingDeviceDriver/Introduction/Intro.html

[30] J. R. Larus and P. N. Hilfinger, "Detecting conflicts between structure accesses," *ACM SIGPLAN Notices*, vol. 23, no. 7, pp. 24–31, 1988.

[31] F. Wang and Y. Shoshitaishvili, "Angr - the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 8–9.

[32] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, vol. 8, 2008, pp. 209–224.

[33] D. A. Ramos and D. Engler, "Under-Constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 49–64.

[34] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[35] "Upgrading to an m1/m2 mac," 2023. [Online]. Available: https://communities.vmware.com/t5/VMware-Fusion-Documents/Running-Fusion-on-an-Intel-Mac-and-upgrading-to-an-M1-M2-Mac/ta-p/2888565

[36] "Linux testing project," 2023. [Online]. Available: https://github.com/linux-test-project/ltp

[37] "Linux kernel selftests," 2023. [Online]. Available: https://www.kernel.org/doc/Documentation/kselftest.txt

[38] "Open posix test suite," 2023. [Online]. Available: https://posixtest.sourceforge.net/

[39] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[40] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[41] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 380–394.

```
struct autofs_dev_ioctl *param;
/* Copy the parameters into kernel space. */
param = copy_dev_ioctl(user);
if (cmd != AUTOFS_DEV_IOCTL_VERSION_CMD &&
    cmd != AUTOFS_DEV_IOCTL_OPENMOUNT_CMD &&
    cmd != AUTOFS_DEV_IOCTL_CLOSEMOUNT_CMD) {

    struct super_block *sb;
    fp = fget(param->ioctlfd);
    ... ...
```

Figure 4: Excerpt from Linux kernel for the driver autofs

Appendix A.

A.1. Dependency in Syzkaller

We observed Syzkaller specifies 31 more dependencies than the ground truth, which we found to be either false positives or for code that is not enabled by its default configuration. They are present in three drivers (*i.e.,* `autofs`, `dri` and `kvm`) due to the following reasons: (1) To ease the development of specifications, Syzkaller supports type templates allowing developers to factor out common fields for similar types and declare derived types through inheritance. Thus, for the driver `autofs`, Syzkaller specifies a base type with a dependency, and all 14 interfaces reuse it to reduce redundancy. However, as illustrated in Fig. 4, we can observe that all commands share the same input structure `struct autofs_dev_ioctl` but there are three exception cases where the specified handler `ioctlfd` is not used (*i.e.,* false positives) despite being declared in the input structure. Syzkaller follows the type definitions from the source code without validating them. (2) As for `dri`, Syzkaller specifies seven dependencies for legacy code that is not enabled by its configuration. (3) Similarly, Syzkaller consolidates specifications for different architectures into one file for KVM. For instance, Syzkaller specifies the following interface which is only available when the kernel is built for a particular architecture (*i.e.,* S390):

```
ioctl$KVM_S390_UCAS_MAP(fd fd_kvmcpu, cmd const[
    KVM_S390_UCAS_MAP], arg ptr[in,
    kvm_s390_ucas_mapping])
```

A.2. False Positives

Our solution did not produce any false positives, indicating the robustness of our approaches, including identifying insertion and lookup operations as dependencies (as discussed in §3.2.3) and the sequential allocation of "handler" (as described in §3.2.4). Interestingly, we found only two drivers, namely `rdma_cm` and `kvm`, have false positives if we simply rely on the first heuristic due to the following reasons:

1) `SyzGen++` found a plausible dependency between two interfaces `rdma_listen` and `rdma_bind_ip`. While `rdma_listen` allocates a port number and creates an associated kernel object that is stored in some data container and can be retrieved through the port number, `rdma_bind_ip` binds to a certain IP address combined with a user-provided port number after it ensures the port number is available. The way it checks the availability of a port number is to look up the data container with it, and it only proceeds when it fails to retrieve an object. Although `rdma_listen` and `rdma_bind_ip` satisfy the requirements of insertion and lookup, respectively, they do not conform to our definition of explicit dependency because the lookup operation is to ensure there is no corresponding object instead of retrieving one created by the insertion operation. Since `rdma_bind_ip` does not reuse the port number produced by `rdma_listen`, `rdma_listen` does not return the newly-assigned port number to userspace. Thus, `SyzGen++` easily rejects this potential dependency based on the second heuristic because `rdma_listen` does not have any output buffer.

2) As mentioned in §3.2.3, `SyzGen++` utilizes field offsets to represent access paths and thus cannot distinguish different subfields within `kvm` as shown below.

```
struct kvm {
  ... ...
  kvm_io_bus *buses[KVM_NR_BUSES];
  ... ...
  hlist_head irq_ack_notifier_list {
    hlist_node *first;
  };
}
Insertion: kvm->irq_ack_notifier_list->first
Lookup:    kvm->buses + index * sizeof(kvm_io_bus)
```

`SyzGen++` does not consider the constraints upon the user input when matching pairs due to the overconstraint issue introduced by symbolic execution [41]. As a result, `SyzGen++` may incorrectly assume that the lookup operation can retrieve more objects beyond the fixed-sized array "buses". Thanks to the second heuristic (*i.e.,* sequential allocation of "handler"), `SyzGen++` could successfully prune this false positive.

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

`SyzGen++` automatically infers dependencies between syscalls to generate specifications for more efficient fuzzing. The paper uses symbolic execution to look for a dependency design pattern (insert and lookup). In a one-month fuzzing campaign, they find four new CVEs depending on these specifications.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) Creates a new tool to enable future science, in case that SyzGen++ would be open-sourced.
2) Addresses a long-known issue, which is the dependencies of syscalls that create a barrier in finding bugs deeper in the Linux kernel.
3) Provides a valuable step forward in an established field, by creating a fuzzer that found new CVEs in the Linux kernel.

B.4. Noteworthy Concerns

1) SyzGen++ cannot handle dependencies of more complex syscalls like mmap.
2) During the one-month fuzzing campaign on the Linux kernel, half of the bugs uncovered were not found due to the SyzGen++ dependency improvements. It is unclear how effective SyzGen++ is in practice, as bugs are the only metric in the paper.
3) The reviewers did not have source code access to SyzGen++. As SyzGen++ has more than 12k lines of code, it could contain a lot of special cases, and the algorithm description indicates it tends to produce a lot of false positives. Applicability to multiple Linux kernel versions was not shown.
4) Unfair comparison with other works, claiming that Moonshine would not find any dependencies.
5) The macOS evaluation is limited. While the authors state that dependencies could also be inferred for macOS, the dependencies identified are way lower than SyzGen's findings. SyzGen found 65 dependencies not found by SyzGen++, while SyzGen++ only found 3 dependencies not found by SyzGen.
6) The paper considers a very limited set of macOS IOKit drivers. With IOKit being open-sourced by Apple, regular releases of Apple's Kernel Development Kits for macOS, various reversing done by the Asahi Linux project, and knowledge about IOKit being documented in Jonathan Levin's books, analyzing more drivers should be possible.
7) The paper did not attempt to fuzz the macOS kernel, even though various drivers are reachable from user space without disabling SIP.

Appendix C. Response to the Meta-Review

We provide our responses below to address those noteworthy concerns one by one.

1) We agree that SyzGen++ cannot handle some complex cases yet and have discussed the limitations of our approach in §5.2 and §6.
2) We added the discussion of the necessity of accurate dependency inference in §5.5.
3) As mentioned in §1, we have opened source our tool at https://github.com/seclab-ucr/SyzGenPlusPlus. As we discussed in §4, we do not target any specific kernel versions, nor handle any special cases. We do model some common functions like kmalloc and free as any other symbolic execution-based solutions, but we don't expect them to change frequently across different versions. As mentioned in §5.2, we didn't find any false positives thanks to our dynamic checker based on the observation of sequential allocation of handlers. Even without the checker, we found only two drivers (i.e., rdma_cm and kvm) have false positives and provided a detailed explanation in Appendix A.2.
4) We have realized that including our test results for Moonshine without sufficient explanation could potentially mislead readers, giving the incorrect impression that Moonshine is entirely ineffective. To rectify this, we have provided a detailed explanation in §5.2. Moonshine, along with any other trace-based solutions, can discover dependencies if valid traces exercising those dependencies are supplied.
5) We explained the reason why SyzGen++ failed to identify those dependencies found by SyzGen in §5.2. This is because those dependencies were handled by the underlying firmware, which is not analyzed by our tool. This is more like a limitation of our current implementation, not the design. We believe it could be addressed with additional engineering efforts.
6) SyzGen++ leverages SyzGen to support macOS, which only supports IOKit drivers and VMWare Fusion-based macOS setup. Additionally, VMWare Fusion does not support macOS 12.3 Monterey or later yet [35]. Hence, we can only test the same old version (macOS 10.15.4) as SyzGen and reused the same dataset of IOKit drivers. Note that the number of IOKit drivers running inside a VM is much less than that of a physical machine and SyzGen excludes drivers that require root access. It is possible to analyze more drivers if we could improve SyzGen to either utilize a physical machine or switch to another VM that supports new macOS versions, though it may require significant engineering efforts. Even if we could achieve it, we do not have the test suites/traces for those new drivers and it would be unfair to conduct the comparison between SyzGen and SyzGen++. We added more details in §5.1.
7) Same as 6. Fuzzing an older macOS kernel is less likely to yield new bugs, particularly considering the limited number of IOKit drivers available within a Virtual Machine that do not require root access. We have clarified it in §5.5.