# SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers

Weiteng Chen
University of California, Riverside
Riverside, California, USA

Yu Wang
Didi Research America
Mountain View, USA

Zheng Zhang
University of California, Riverside
Riverside, California, USA

Zhiyun Qian
University of California, Riverside
Riverside, California, USA

## ABSTRACT

Kernel drivers are a critical part of the attack surface since they constitute a large fraction of kernel codebase and oftentimes lack proper vetting, especially for those closed-source ones. Unfortunately, the complex input structure and unknown relationships/dependencies among interfaces make them very challenging to understand. Thus, security analysts primarily rely on manual audit for interface recovery to generate meaningful fuzzing test cases. In this paper, we present SyzGen, a first attempt to automate the generation of syscall specifications for closed-source macOS drivers and facilitate interface-aware fuzzing. We leverage two insights to overcome the challenges of binary analysis: (1) iterative refinement of syscall knowledge and (2) extraction and extrapolation of dependencies from a small number of execution traces. We evaluated our approach on 25 targets. The results show that SyzGen can effectively produce high-quality specifications, leading to 34 bugs, including one that attackers can exploit to escalate privilege, and 2 CVEs to date.

## CCS CONCEPTS

• **Security and privacy** → **Vulnerability scanners**; **Operating systems security**.

## KEYWORDS

fuzzing, operating system security, vulnerability analysis

**ACM Reference Format:**
Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. 2021. SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3460120.3484564

## 1 INTRODUCTION

According to syzbot [5], Google's Linux kernel fuzzing platform, 2,854 bugs have been found in the Linux upstream kernel in just under four years of deployment. This translates to 3 bugs per day on average, demonstrating the tremendous success of its underlying kernel fuzzing system, namely Syzkaller [25]. More importantly, according to the report [24] from Google and some prior research[30], the majority of Linux bugs reported are attributed to drivers as they contribute to a large portion of the codebase and oftentimes are less tested, indicating a critical attacking surface. It is no different from Apple's operating systems. There were 74 CVEs related to Apple drivers, accounting for approximately one-third of all 231 reported Apple kernel vulnerabilities from iOS 8 through iOS 13.4.1 [7].

The key to the success of kernel fuzzing hinges on a fuzzer's ability to generate diverse and interesting test cases that exercise various corner cases relatively deep in the kernel. Today, this is largely accomplished through syscall specifications that are typically manually crafted by experts. For example, Syzkaller, the state-of-the-art kernel fuzzer, supports templates that encode the information regarding syscalls that can be invoked against specific kernel modules. More specifically, they contain two types of information about syscalls: (1) The structures and constraints of syscall arguments, *i.e.,* type, value ranges, and the relationship between fields. Without such knowledge, the input generated by a fuzzer will likely be rejected by the kernel as driver-specific sanitization will be performed on untrusted input from userspace. (2) `Dependencies` between syscalls. This is crucial because a kernel module maintains its internal states: successful execution of syscalls usually require the right sequence of invocation (*i.e.,* `implicit dependence` or `ordering dependence`) and/or correctly passing a 'handler' (*e.g.,* file descriptor) returned from the kernel to a syscall (*i.e.,* `explicit dependence` or `value dependence`) [14]. Missing explicit dependencies can be especially detrimental because key functionalities of a kernel module would become unreachable, *i.e.,* it is unlikely a fuzzer can generate a random value that happens to match a specific 'handler' returned by previous syscalls.

Unfortunately, the process of curating templates is tedious and labor-intensive, often requiring a deep understanding of the corresponding module. As a result, in practice, templates are incomplete and lead to sub-optimal fuzzing results. Indeed, from tracking the history of templates maintained by Syzkaller [6] over the years, there are a large number of additions and corrections to improve the quality.

Despite the challenge, there has been recent work on automating the generation of syscall templates. Specifically, DIFUZE [9] was proposed to statically analyze the source code of a Linux kernel module to infer the structure and constraints of syscall arguments, based on how the arguments are copied and used in the module. In


*CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea*

ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3484564

addition, it also compiles a list of hard-coded explicit dependencies in kernel modules.

In this project, we take on an ambitious goal to automatically generate syscall templates for *closed-source* drivers on macOS. There are several unique challenges in achieving the goal. First, unlike the core kernel's syscalls which are well-documented to support application development, drivers' syscalls are generic and yet have vastly different functionalities depending on the underlying driver-specific implementation. For example, `IOConnectCallMethod` in macOS (or its counterpart `ioctl` in Linux) is a generic syscall that takes a `void*` argument to communicate with any driver. Second, since we target closed-source drivers, it is much more difficult to recover information regarding syscall arguments and dependencies among syscalls (*e.g.,* lack of type, inlined functions). This also means that we cannot directly apply the recent work [9] that statically analyzes the source code of Linux kernel modules to automated specification generation.

To overcome the challenges, we present `SyzGen`, driven by two key insights: (1) Iterative refinement. Templates can be generated and refined over time instead of being curated in one shot. This allows us to overcome the challenge of having to precisely analyze the whole binary-only driver. Instead, we can sample various execution paths and combine the knowledge from each. (2) Explicit dependencies can be extracted and *extrapolated* based on a small number of execution traces. This allows us to map out the explicit dependencies that we may not have seen in the past, creating much more complete templates.

`SyzGen` is the first to automate the generation of syscall specifications for closed-source macOS drivers and facilitate interface-aware fuzzing. We evaluated our tool against 25 targets without source code and discovered 34 bugs, 2 of which have been assigned CVE numbers so far. We also observed that `SyzGen` could identify 271 explicit dependencies and produce high-quality specifications by measuring the code coverage, demonstrating the effectiveness of our explicit dependence inference and interface recovery.

In summary, we make the following contributions:

- **Interface-aware fuzzing of binary-only drivers**. We developed `SyzGen` capable of automatically extracting both structures/-constraints of syscalls and explicit dependencies between syscalls, given a specific macOS driver. We released the source code of our prototype to facilitate the reproduction of results and future research: https://github.com/seclab-ucr/SyzGen_setup.
- **Novel techniques**. We leveraged two insights to get around the challenges in binary analysis: (1) iterative refinement of syscall knowledge and (2) extraction and extrapolation of explicit dependencies from a small number of execution traces.
- **Promising experimental results**. We evaluated `SyzGen` against 25 targets on macOS and found 34 bugs, 2 of which have been assigned CVE numbers so far.

## 2 BACKGROUND AND RELATED WORK

In this section, we will give some brief background on the internal structure of macOS drivers, which are the main targets of this paper, and introduce prior work to explain the challenges we must overcome.

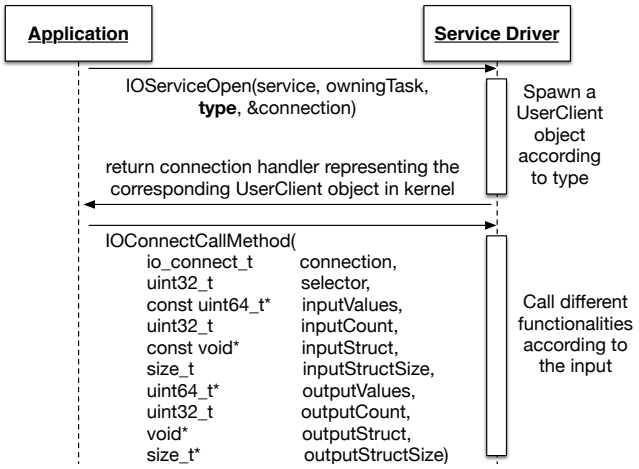



**Figure 1: The internal structure of drivers and its interface.**

### 2.1 MacOS Device Drivers

Similar to Linux, MacOS provides a few generic syscalls such as `IOServiceOpen` and `IOConnectCallMethod` through which a user-space application can interact with a driver. Specifically, each driver can expose a few services through specific service names (hard-coded strings), each of which in turn can have a few user clients providing different functionalities. Note that user clients reside in kernel space and are part of the drivers. Fig. 1 depicts the typical communication process for an application to interact with a kernel driver. Any user application that wishes to connect to a service must firstly invoke `IOServiceOpen` (first argument `service` specifies which). Then the second argument `type`, an unsigned 32-bit integer, is interpreted by the service [1] to instantiate a corresponding user client object responsible for subsequent communication between the user application and service. Upon a successful invocation, a connection handler is returned to the caller which can be used to locate the user client object in kernel. Any request that the application sends to the driver will be made by calling `IOConnectCallMethod()` (the main syscall) that takes this connection as the first parameter.

As shown in Fig. 1, `IOConnectCallMethod()` is a generic syscall that can take any complex data structures, and is implemented differently by each driver. The second parameter `selector` is commonly known as the "command identifier" to determine which operation the user client would perform. The rest eight parameters are used to pass inputs and gather outputs. Specifically, `inputValues` contains integer-only inputs and `inputCount` determines the number of elements in `inputValues`. In contrast, `inputStruct` can contain arbitrary types of inputs (as its type is `void*`), and its size is specified in `inputStructSize`. The four output parameters are similar in nature compared to the input ones. We refer to the collection of functionalities corresponding to a specific command identifier value as an **`interface`**. The separation of interfaces follows the convention of Syzkaller templates and allows the different interfaces to be treated differently as will be shown later.

[1] It may be omitted by the driver if there is only one user client to serve.

There are a few interesting things we wish to point out. First, to conduct any meaningful fuzzing, it is critical to infer the mapping for each interface between the value of a command identifier and the rest of the arguments (input or output). Second, even though there is a convention, drivers can interpret the inputs in any way they choose, and the command identifier does not necessarily need to be passed as the second argument (*i.e.,* `selector`). It can be embedded in one of the parameters labeled as inputs. Finally, interfaces can have dependencies on each other. New and interesting code can be revealed only when a test case exercises a dependence by invoking multiple interfaces with the correct arguments.

## 2.2 Kernel Fuzzing

Coverage-guided fuzzing [32] is now the de facto standard for testing and bug finding in the industry due to its efficacy in discovering complex vulnerabilities without false positives. The unique aspect of kernel fuzzing is that the input is comprised of a sequence of syscalls, involving complex arguments and dependencies. To address the challenge, the state-of-the-art kernel fuzzer Syzkaller [25] was developed to allow developers to encode the knowledge of syscalls in the form of templates.

Below we summarize the recent efforts on interface recovery assisting kernel fuzzing. Specifically, we list their characteristics in Table 1.

**Interface Recovery without Dependence Inference**. DIFUZE [9], dedicated to recovering interfaces of Linux drivers, is the most relevant work. It conducts a static analysis to retrieve command identifiers and their corresponding input structures. However, it requires source code to extract the structure definition and thus can not be applied to closed-source kernel modules including macOS drivers. Moreover, it only conducts static range analysis to a certain argument (*i.e.,* `ioctl's command identifier`) to refine the syscall templates. It also fails to extract complex relationships between fields of structures (*e.g.,* a length field specifies the size of a buffer) and dependencies between syscalls (other than a hard-coded list), which impedes the fuzzer from exploring deeper and more interesting code.

Regarding closed-source macOS drivers, there is a small tool, p-joker [28], which was developed in the industry [3] to recover the interface of some drivers. The idea is based on some common programming pattern in macOS where command identifiers are often used as indices into *function dispatch tables* to locate the corresponding handler function. Following Apple's guidelines, developers can encode the required values for `inputCount`, `inputStructSize`, `outputCount`, and `outputStructSize` in such dispatch table, which would be enforced by the kernel. p-joker also extracts the information to facilitate fuzzing. Unfortunately, this is not the only way a command identifier is used. In addition, the tool is unable to recover types and other constraints of the arguments associated with the command identifier.

**Interface Recovery with Dependence Inference**. As opposed to structure recovery, IMF [14] works on general syscall interfaces of which the argument types are well documented. IMF rather focuses on mutating the value of arguments in a black-box manner, without an understanding of their valid ranges and does not attempt to generate syscall templates. In addition, it also attempts to infer the dependence between syscalls by analyzing existing syscall traces generated by applications. Intuitively, it preserves the order of syscall sequences to produce a fuzzing harness, and infer the explicit dependence by checking the identical value pairs from the input and output of syscalls. Similarly, Moonshine [21] relies on traces to infer explicit dependence and implicit dependence. However, both schemes cannot be directly applied to macOS drivers where the interface argument type is generic (`void*`). Furthermore, none of the approaches attempts to extrapolate dependencies beyond the traces that have been observed, and thus the quality of the inferred dependencies is heavily dependent on the applications that may exercise various functionalities of the corresponding kernel module to various degrees.

A more recent work dubbed HFL [16], a hybrid Linux kernel fuzzer, employs concolic execution to monitor every possible read and write pairs along the execution of a sequence of syscalls in a given test case to find dependencies, which is unfortunately not very scalable and challenging in practice because it needs to drive the execution perfectly to exercise both of the read and write. In contrast, `SyzGen` is much more realistic as it requires the analysis of a single interface only by generalizing the knowledge gathered from prior dependencies (see §4.3). In addition, HFL requires source code to conduct static analysis for instrumentation and points-to relationship, and such analysis is much less precise on binaries.

**Type Recovery**. To support fuzzing, type inference is necessary but not a strong requirement. For example, it is important to differentiate a pointer from non-pointer types, and string (char array) from other array types. However, it is not critical to differentiate unsigned from signed integer, as long as we know what value is interesting and allows more coverage. As a result, we borrow ideas from the rich literature on reverse engineering of variable types in binary programs [10, 17, 19]. Tupni [10] leverages dynamic analysis to recover input formats based on the usage of input. REWARDS [19] proposes to propagate type information based on "type sinks", which are calls to functions with known type signatures (*e.g.,* a library call). In contrast, TIE [17], a static analysis based approach, proposes a principled type inference system that could generate type constraints based upon how the binary code is used and then deduce the actual types. Our type recovery method is similar to Tupni [10] but is simpler due to the lower requirement.

**Other kernel fuzzing work.** In addition to the above, we have seen several other related works in recent years. Moonshine [21] improves syzkaller by distilling seeds of high quality from existing testing suites. JANUS [31], specific to fuzz file system, extends the attacking surface to disk image of which metadata could be malicious and thus leads to vulnerabilities that are neglected by other fuzzers. Similarly, PeriScope [23] is tailored to detect driver vulnerabilities reachable from the hardware side as opposed to the syscall side. Razzer [15] and KRACE [29] combines static analysis and fuzzing to drive fuzzer towards most potential spots of data race bugs. Though these techniques prove to be effective, one of the fundamental reasons for their success is the interface specifications that are manually implemented by security analysts, which is a tedious process given the massive amount of driver code in kernel. What's worse, if the source code is not available, analysts usually

| Tool | Target | Requirements | | | Techniques | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Source Code | Trace | Specification | Infer Explicit Dependence | Infer Implicit Dependence | Coverage Guided | Structure Recovery | Constraint Recovery |
| DIFUZE[9] | Android Driver | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| HFL[16] | Linux Driver | ✓ | ✗ | ✗ | SE | ✗ | ✓ | ✓ | ✓ |
| Moonshine[21] | Linux | ✓ | ✓ | ✓ | DM | ✓ | ✓ | ✗ | ✗ |
| p-joker[28] | MacOS Driver | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| IMF[14] | MacOS | ✗ | ✓ | ✓ | DM | ✓ | ✗ | ✗ | ✗ |
| **SyzGen** | MacOS Driver | ✗ | ✓ | ✗ | DM+SM | ✓ | ✓ | ✓ | ✓ |

**SE**: Symbolic execution on multiple syscalls. **DM**: Data mining on traces. **SM**: Signature matching.

**Table 1: The comparison of recent fuzzing techniques on interface recovery.**

resort to reverse engineering to recover the interface, which is time-consuming and error-prone.

## 3 OVERVIEW

In this section, we first walk through a motivating example to demonstrate our key observation from security analysts' experience on how to infer explicit dependence and refine templates iteratively — which is crucial for developing specifications of good quality, then position SyzGen in a bigger picture.

### 3.1 A Motivating Example

Fig. 2 presents some code excerpts adapted from the macOS driver AppleUpstreamUserClientDriver in which both functions `CloseLink()` and `FlushLink()` require an identifier returned from `OpenLink()` (*i.e.,* two explicit dependencies). Here we consider `OpenLink()` a *generate* interface as it generates a new kernel object and returns a corresponding id (which we refer to as *dependence variable*). In contrast, we consider `CloseLink()` and `FlushLink()` *use* interfaces, as they rely on or "use" the object previously generated.

In this example, we are able to observe the execution traces of `OpenLink()` and `CloseLink()`. Assuming we already successfully inferred the types and constraints of the arguments for both `OpenLink()` and `CloseLink()`, we can also infer the explicit dependence following the prior approach [14]. Specifically, as shown in ❶ in Fig. 3, it is clear that the actual return value of the *generate* interface `OpenLink()` and the value of the first argument of the *use* interface `CloseLink()` always match. This will allow us to generate an initial template involving all three interfaces of `OpenLink()`, `CloseLink()`, and `FlushLink()` but only one explicit dependence is established. This is because we never observe any traces involving `FlushLink()` in a dependence with `OpenLink()`.

Nevertheless, during the course of analyzing `CloseLink()`, we can extract more details about the dependence to help generalize it to other interfaces such as `FlushLink()`. Specifically, we observe that there is a function `LookupLink()` in Fig. 2 responsible for converting a dependence variable (*i.e.,* LinkID) into a corresponding kernel object. This allows us to label `FlushLink()` as an internal dependence operation (see ❷ in Fig. 3) and look for similar operations in other *use* interfaces (note that an internal dependence operation does not have to be a function invocation). The next time we encounter the same internal operation (*i.e.,* `LookupLink()` invocation) in another interface (*e.g.,* `FlushLink()`), we can conclude the passed value is a dependence variable of the same type, *i.e.,* `LinkID` (see ❸ in Fig. 3). We can further observe that `LinkID` comes

```
01 typedef int32 LinkID
02 struct CloseRequest { LinkID linkID; };
03 struct FlushRequest { unknownFields };
04 Service* gService; // Struct definition is omitted.
05 int OpenLink() { … … return linkID; }

06 void* LookupLink(int linkID) {
07     … …
08 }
09 int CloseLink(struct CloseRequest* arg) {
10     p ← LookupLink(arg->linkID);
11     if (p == NULL) goto error;
12     … …
13 }
14 int FlushLink(struct FlushRequest* arg) {
15     p ← LookupLink(*(int*)&arg->unknownFields[0]);
16     if (p == NULL) goto error;
17     magic ← *(int*) &arg->unknownFields[1];
18     if magic != 0xdeadbeef: goto error;
19     … …
20 }
```

**Figure 2: A motivating example for explicit dependence inference. If we know CloseLink accepts a dependence LinkID, we can also learn that FlushLink requires the same LinkID due to their similar code pattern.**

from the four bytes of the `arg` of `FlushLink()`, and therefore conclude `FlushLink()` is dependent on `OpenLink()` and update the template with the new dependence accordingly.

Next, we can inspect other fields in `arg` of `FlushLink()` and refine the template even further with the types and constraints regarding the complete `arg` (see ❹ in Fig. 3). For example, we may learn that the second field of `arg` needs to take a magic number to reach a deeper part of the function. The process of iterative refinement of specifications, starting from a "sampled" execution paths (including `OpenLink()` and `CloseLink()`, allows us to gather a progressively more complete understanding of the driver. We argue that this side-steps the challenge of analyzing a complex driver in binary as a whole, and is also a suitable process for automation.

### 3.2 System Architecture

Fig. 4 illustrates the system architecture of SyzGen which is aimed at generating specifications for macOS drivers with respect to dependencies between interfaces. SyzGen primarily consists of four components including (1) syscall logger and analyzer, (2) service

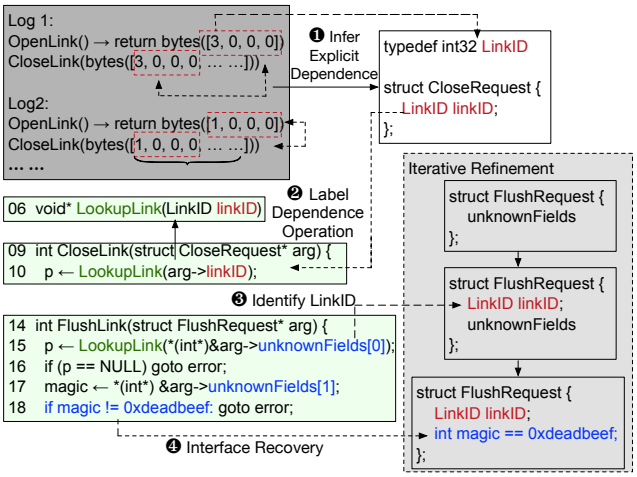



**Figure 3: Typical process of interface recovery for the motivating example. ❶ Inferring explicit dependence by searching for identical input and out pairs from logs; ❷ Annotating dependence related operations; ❸ Identifying more dependence based on annotated code; ❹ Recovering structure and constraints of inputs.**

and interface identification, (3) interface recovery, and (4) fuzzer with coverage enabled.

**Syscall logger and analyzer**. The logger instruments the kernel to record the input and output of every syscall to the target driver. And then SyzGen analyzes the collected logs to identify explicit dependencies that are directly observable in the execution traces (following the approach in IMF [14]), which may be limited as mentioned earlier. It further separates the logs into independent test cases according to the dependence we have inferred to produce an initial corpus.

**Service and command identifier determination**. Given the target binary, SyzGen detects the service name and its type number (corresponding to user clients), which are used to interact with the driver. As mentioned in section §2.1, since interfaces share the same entry, *i.e.,* `IOConnectCallMethod()`, SyzGen also needs to find out what `command identifier` values the driver expects and figure out where it is in the input so that the syscall analyzer could distinguish different interfaces from each other.

**Interface recovery**. For each interface, SyzGen first attempts to generate an initial template encoding the previously extracted explicit dependence knowledge, along with the input structure and constraints through dynamic analysis (on sampled execution paths). In addition, it attempts to automatically extrapolate or generalize the explicit dependence from known ones, in a style similar to the motivating example. Then, SyzGen proceeds iteratively, allowing it to gradually encode newly-discovered dependencies and refine the structure and constraints of new *use* interfaces (*e.g.,* `FlushLink()`).

**Fuzzer with kernel coverage**. Given the specification SyzGen produces, a standard Syzkaller can start the fuzzing campaign. However, as much of its power comes from the fact that it is coverage-guided, we also integrate a kernel module responsible for collecting coverage in the system, which does not require the source code or

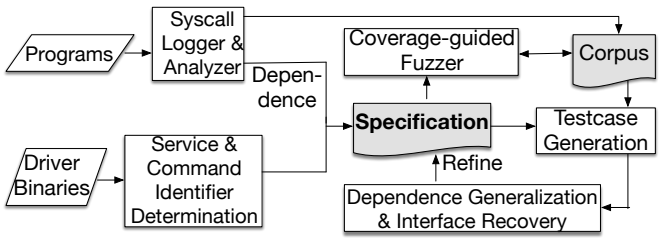



**Figure 4: Workflow of SyzGen**

specific hardware and virtual machine. With the coverage, SyzGen additionally infers implicit dependence (order of syscalls) to further improve the effectiveness of test case generation.

Although in this work we focus on macOS, the proposed solution can be applied to other OSes with closed-source kernel modules. It is also worth noting that if no traces are available, SyzGen can still work, though it can no longer infer explicit dependencies from existing traces and extrapolate them.

## 4 DESIGN

In this section, we describe the design of SyzGen in depth. For each component we present in §3.2, we will explain our design decisions.

### 4.1 Syscall Logger and Analyzer

As mentioned in §3.2, the primary goal of this component is to extract the explicit dependence from existing traces. Together with the service detection and command identifier determination, we will be able to generate preliminary versions of templates. Though we apply a similar idea as IMF [14], we also address its limitations. Specifically, the fundamental premise of IMF is the availability of knowledge of specifications for the target syscalls (including parameter definitions), which does not hold true for drivers. As mentioned in §2.1, the syscall `IOConnectCallMethod()`'s key input and output parameters `inputStruct` and `outputStruct` are `void*`. Even though their sizes are given by `inputStructSize` and `outputStructSize`, it is unable to discern which field of `inputStruct` represents a dependency. Furthermore, according to our finding, it is common that `inputStruct` contains pointers that point to other objects (this can go recursively also). As a result, IMF is unable to track explicit dependencies whenever the dependence variable is located in the `void*` object.

In our solution, we handle the `void*` by assuming the dependence variable can exist anywhere in the object, from a single byte to at most eight bytes. Basically, we search for pairs of identical input and output bytes in two different interfaces. If there exists a single byte whose values in both the input and output always match, we consider this byte a potential dependence variable. However, when there are multiple such individual bytes, we merge them into contiguous groups up to eight bytes long. Interestingly, in practice, we find that the size of dependence variables can indeed vary from two to eight bytes. To improve precision, we ensure identified explicit dependencies are consistent across different logs and exhibit different values. The rationale behind it is that dependence

value by nature is a dynamically changing resource whose value is inconstant if we have seen enough logs.

In addition, to reconstruct the additional objects that are reachable by pointers in `inputStruct` (which may contain the dependence variable), we monitor the key internal APIs invoked in macOS drivers to transfer the input from userspace to kernel space, *e.g.,* the macOS equivalent of `copy_from_user()` in Linux. For example, if there is only one pointer in `inputStruct` object, there will be one copy of the `inputStruct` itself, and then a second copy of the data at the address given by the pointer. We perform the reconstruction recursively by following as many layers of pointers as necessary. Combining the above two improvements, we are able to locate a more complete set of dependence variables and extract more explicit dependencies missed by IMF.

Finally, a secondary goal of this component is to generate concise test cases that encode explicit dependencies, of which the benefits are twofold: (1) The later dynamic analysis in interface recovery is performed against these distilled test cases instead of the entire log which can be expensive to sift through. (2) Such test cases can serve as the initial corpus to boost the fuzzing campaign.

## 4.2 Service and Command Identifier Determination

To generate a complete template, `SyzGen` needs to know the service names (specific strings) and valid values of service types to determine the exposed services and user clients (see §2.1 for details). In addition, we need to infer the valid values for the `command identifier` to differentiate different interfaces.

**Service identification**. By design, the name of a service is the name of the corresponding class. As a result, we can directly resort to the symbols indicating the service class names. Next, we can simply query the OS using a convenient API `IOServiceMatching()` with the service name to confirm its validity. This gives us a list of matching services registered in the system. To infer the valid values of service type, for each service, we conduct a dynamic symbolic execution of `IOServiceOpen()` with the third argument `type` symbolized (See §5 for details.).

**Command identifier determination**. As mentioned earlier, command identifiers may or may not be passed as the second argument of `IOConnectCallMethod()`. When they are not, it can be tricky to determine which bytes in either `inputValues` or `inputStruct` correspond to a command identifier. Nevertheless, our observation is that a command identifier is used to determine which functionality the service should provide, and there are generally some common programming patterns in macOS. As mentioned in §2, there is often a function dispatch table that takes the command identifier as an index to invoke different functions representing different functionalities. However, using the command identifier as an index may not be the only pattern (which was recognized by p-joker [28]). We find that it can also be implemented by involving the command identifier in conditional statements, *e.g.,* switch cases or if-else statements, to determine the subsequent code to execute.

Given the above, we design a general symbolic-execution-based exploration strategy to identify such patterns. Basically, we attempt to find a symbolic variable (among all the symbolized ones) whose values lead directly to different functions being invoked. For example, we may find a symbolic variable `a` whose constraint is `a == 1` when function `foo()` is invoked, whereas it has a different constraint `a == 2` when function `bar()` is invoked. The exact algorithm is described in Algorithm 1 in the Appendix. In practice, we find that the algorithm is general enough to handle a variety of patterns mentioned above (See Appendix A for more details). Note that the symbolic variable identified by the algorithm will not only tell which parameter the `command identifier` comes from (*e.g.,* the fifth argument `inputStruct` of `IOConnectCallMethod()`) but also precisely which bytes (*e.g.,* first 8 bytes of `inputStruct` or the object pointed to by a pointer field of `inputStruct`).

After identifying the service and command identifier, `SyzGen` could generate an initial template. Moreover, as described in §2, some macOS drivers follow the convention of using function dispatch tables which also encodes some basic constraints enforced by kernel automatically (*i.e.,* desired sizes of the input and output). Similar to p-joker [28], `SyzGen` extracts such constraints whenever available (typically common in simple drivers) and encodes such information in the initial template as well. In the next component, we will describe the more fine-grained structure and constraint inference.

## 4.3 Interface Recovery

This is a core part of `SyzGen`, which aims at reconstructing the argument structures and collecting their more fine-grained constraints. Most importantly, `SyzGen` generalizes the knowledge of dependencies it has learned from logs (see §4.1) to those interfaces without any trace and thus can uncover more dependencies.

**Choice of dynamic symbolic execution**. A recent work DIFUZE [9] has opted for static analysis to reconstruct the nested argument structures from source code. This is a reasonable choice when the source code is available. Even then, static analysis is limited due to the challenge of precisely reasoning about program values and pointer relationships. It gets even worse for closed-source macOS drivers. As a result, we choose dynamic symbolic execution instead, which fits our problem better. First, analyzing the interfaces dynamically (with concrete memory states) allows us to bypass the precision challenge of static analysis. Second, we are able to collect useful constraints about valid ranges of various arguments as well as relationships among different fields. However, the downside is that it may suffer from the path explosion problem. Fortunately, in the case of drivers, syscall arguments are usually checked at the very beginning of syscalls (most are simple sanity checks) and thus `SyzGen` only needs to perform symbolic execution up to these points, which is much more manageable.

**Test Case Generation.** To perform dynamic symbolic execution, we need to obtain valid test cases that correctly set up the context (*e.g.,* global variables) so we can symbolize the arguments of an interface to explore deeper parts of the code. As mentioned in §4.1, we have an initial corpus of test cases that already exercise known explicit dependencies. Going back to the motivating example, we can easily obtain a test case with both `OpenLink()` and `CloseLink()`, exhibiting a dependency. Therefore we can symbolize the argument of `CloseLink()` and learn the structure and constraints of it, without worrying about an early exit of the function due to the lack of

a correct dependence variable. We also allow the fuzzer to generate a few more variants to improve the diversity as explained in §5.

For the cases where we do not have any valid test case for an interface, we will need to do some more work iteratively. For the same motivating example, since we do not have any test case that has exercised `FlushLink()`, SyzGen will first try to infer any potential explicit dependence with other interfaces (see later in the section). At a high level, SyzGen will initially generate a test case exercising only one interface (in addition to the prologue of `IOServiceOpen()`), and it will then use the learned knowledge about its arguments and any new explicit dependence to continue to improve the test case. For the interface `FlushLink()` as an example, though SyzGen would initially fail to explore any deep code with the initial test case as it does not set up proper context and thus leads to an error path. Specifically, it is likely that no `OpenLink()` is invoked, and even if it is, its return value is not passed to `FlushLink()`. Even though we may not be able to learn the structure or constraints of its argument, a symbolic execution still likely allows us to reach the critical function `LookupLink()` responsible for checking the dependence variable, triggering SyzGen to become aware of the potential dependence. After it learns `FlushLink()` also requires a dependence variable of type `LinkID`, it could refine the specification and produce another test case respecting the dependence as follows:

```
int id = OpenLink();
struct FlushRequest req = { .linkID = id };
FlushLink(&req);
```

With the valid sequence of syscalls, SyzGen can then redo the dynamic symbolic execution on `FlushLink()`, extracting more complete knowledge about the structure and constraints of the argument.

**Dependence Generalization.** Now we describe the methodology to generalize dependencies beyond the ones we observed in the past. Using the motivating example in §3.1, given the knowledge of the dependence variable `LinkID` learned from logs (see §4.1), SyzGen first analyzes the *use* interface `CloseLink()` and figures out what internal dependence operation is performed to retrieve the corresponding kernel object generated before. In Fig. 2, it happens through another function call of `LookupLink()`. To recognize such function calls, we observe that the function call must take in the dependence variable as an argument, and return an object. Furthermore, to avoid false dependence being identified due to any irrelevant "helper" functions (*e.g.,* `copy_from_user` liked functions) we allow only the functions defined within the target driver (as opposed to external modules). Our observation for macOS is that external functions in the core kernel (invoked by drivers) are not designed to handle dependencies in drivers.

After discovering such a function call, we label the internal dependence operation to be a function call (`LookupLink()`) together with the parameter corresponding to the dependence variable (1st argument of `LookupLink()`). When symbolically executing a new interface (*e.g.,* `FlushLink()`), we will look for the same internal dependence operation. If there is a match, we will further look at whether the argument of the internal dependence operation is a symbolic variable. If so, it confirms the generalized dependence,

```
01 int CloseLink(struct CloseRequest* arg) {
02     p ← gService->links->head;
03     while (p->value != arg->linkID) {
04         p ← p->next;
05         if (p == NULL) goto error;
06     }
07     … …
08 }
09 int FlushLink(struct FlushRequest* arg) {
10     p ← gService->links->head;
11     while (p->value != *(int*)&arg->unknownFields[0]) {
12         p ← p->next;
13         if (p == NULL) goto error;
14     }
15     … …
16 }
```

**Figure 5: Dependence inference through common access pattern gService→links→head→value if LookupLink is inlined in the motivating example.**

and we can also learn which bytes of the input constitute a dependence variable. This allows SyzGen to encode the new dependence in the template, setting the stage for the next step of structure and constraint recovery.

In practice though, such "lookup" functions may not be implemented as an actual function invocation, or they may have been inlined such that it is hard to recognize in the binary. To support such internal dependence operations, we observe that there must be some form of check to validate the dependence variable and subsequently use it to retrieve the corresponding kernel object. An example is shown in Fig. 5, where a linked list is traversed and the dependence variable is compared against the same field in every element. Another common case is to use the dependence variable as an array index to obtain the corresponding object, for which there is always a check against the index to ensure no out-of-bound access would occur. Based on this observation, we may craft simple signatures based on such checks and match them in new interfaces. However, if we only look at a simple check against the dependence variable, it may lead to false positives. This is because in the new interface (*e.g.,* `FlushLink()`), we do not yet know which bytes correspond to the dependence variable. Therefore, there may be similar checks performed against bytes that are not the dependence variable. Nevertheless, if we carefully examine the check in the example "`p→value != arg→linkID`", we can find that the left side of the check is a value derived from a pointer, which in turn is originated from the global variable "gService" through a chain of dereferences. Intuitively, we can annotate the check with sufficient history (*i.e.,* the origin of the variable) so that the generated signature could be unique enough. Formally, the signature can be formulated in the form of ASTs (*i.e.,* symbolic expressions) following the notation in Fig. 6. For the motivating example in Fig. 5, the corresponding signatures are the following: "`neq [[[gService+264]+8]+8] linkID`" and "`neq [[[gService+264]+8]+8] unknownFields[0]`" [2] where

[2]We omit some structure definitions in the motivating example and those immediate values are offsets to some fields.

```
var:   symbolic variable
imm:   immediate value
[ ]:   dereference
op1:   binary operators
op2:   unary operators
expr:  var | imm | [expr] | op1 expr expr | op2 expr
       | if (expr) then expr else expr
```

**Figure 6: Notation for formula (signature)**

neq means inequality. By simply comparing the two signatures, we can learn that unknownFields[0] is the same dependence variable as LinkID. In addition to this exact match, we also allow a relaxed version of match in which opposite operators can match with each other (*e.g.,* equality and inequality) to cope with some nuances potentially induced by compilers. To further reduce false positives, a valid signature requires at least one dereference and exactly one symbolic variable because it is unlikely the validation of a dependence variable involves other inputs. It is worth noting that our scheme is to mechanically extract formulas (as signatures) from whatever checks performed on the dependence variable and the two types of checks aforementioned are only examples that are correctly identified by our solution.

**Structure and Constraints Recovery.** From the previous steps, we can always have a test case exhibiting a valid explicit dependence, distilled from existing traces or obtained from the dependence generalization. For the test cases that come from existing traces, we already know the rough structure of the `void*` object, including its size, and any additional layers of objects reachable through its pointers from the earlier steps. For the test cases that come from the dependence generalization, we may have the knowledge of the dependence variable but nothing else regarding the input.

The process is slightly dependent on which case we are faced with. In the first case, even though we know the size of the object, up to this point, we still treat each layer of the object as a flat array. To infer more structures, during the symbolic execution, we simply symbolize all the memory associated with the object (including all the layers). We then monitor every "use" instruction of the symbolic memory to determine the boundary of the various fields. Specifically, SyzGen identifies fields of sizes 8, 16, 32, and 64 bits at byte granularity. In addition, we infer the basic types of the fields based on how they are used. The list of supported types is shown in Figure 7. We omit the details as it is following a similar solution to Tupni [10].

In the second case, since we do not yet know the size of the `void*` object, SyzGen initially symbolizes the `void*` input as a flat array with 4,096 bytes. This is because the symbolic length of arrays is poorly handled in symbolic execution engines [4] and thus we start with a size large enough for most inputs. We then perform the same analysis as above to determine the boundary of fields and their basic types. In addition, if any pointer is found (via deference instructions), we will concrete its value to a user-space address, and symbolize the memory accordingly. To determine the size of the symbolic memory, we again look for the macOS-equivalent API `copy_from_user()` as we did in §4.1.

Finally, since our solution is based on symbolic execution, SyzGen generates one template for each explored path and later merges them. In particular, we are interested in retaining the templates for which we are able to explore relatively deeper parts of the kernel. Thus, we prune the templates with paths that terminate early, *e.g.,* due to failing to pass sanity checks. SyzGen applies the hierarchical agglomerative clustering algorithm [11] to group templates that are similar in size, and prune the clusters that correspond to the shorter paths. In particular, the algorithm clusters the templates in the form of a binary tree where the leaf nodes are the individual templates. Our policy is such that if a non-leaf node (corresponding to a cluster) whose centroid of path depth is less than 0.5 of that of the sibling node, we will prune it. To safeguard the shorter but also functional paths (preventing them from being pruned), we also keep the templates whose number of executed basic blocks exceeds a pre-determined threshold (*e.g.,* 500 in our experiments). The parameters of 0.5 and 500 are empirically determined based on the number and quality of generated templates. Lowering those thresholds would preserve more corner paths at the cost of fuzzing efficiency as it increases the search space. In addition to path pruning, SyzGen recursively merges templates as we will explain in §5.

## 5 IMPLEMENTATION

We have implemented SyzGen with 7.2K lines of Python code for interface recovery, 1K lines of C code for kernel coverage, 463 lines of C code for syscall logger, and 1K lines of Go code into Syzkaller for fuzzer. We also implemented scripts based on IDA Pro [2] to collect addresses of basic blocks and function signatures (*i.e.,* the number of parameters and where they are stored).

**Symbolic execution**. Currently, there is no publicly available tool that can perform dynamic symbolic execution of the whole macOS kernel, as what S2E [8] can do on Linux kernels. Fortunately, as articulated earlier in §4.3 , SyzGen only needs to focus on one interface at a time and perform dynamic symbolic execution on a small portion of the driver. As a result, we developed our symbolic execution component based on angr [27] and kernel debugging, allowing us to take a snapshot at any kernel address and prepare a memory state for dynamic symbolic execution. More specifically, SyzGen prepares a test case containing the target interface to set up the proper context (see §4.3), pause the kernel execution when it reaches the target interface, and then conduct symbolic execution under this context (*i.e.,* with the memory snapshot). To improve the scalability of symbolic execution on kernel and cope with kernel functions requiring hardware or multi-threading support, we manually model some kernel functions belonging to the core kernel to be general, such as `strcpy()`, `malloc()`. For the rare cases where driver-specific functions also need modeling (e.g., interacting with hardware), we simply terminate the symbolic execution. Fortunately, such functions are typically behind the input sanity checks, posing minimal impact on constraint extraction. In total, we have modeled 60 functions, 30 of which can be simply replaced with a dummy function, *e.g.,* `printf()` and `sleep()`. Also, we set a 5-minute timeout for each run of symbolic execution since SyzGen only needs to perform symbolic execution to pass sanitization that is usually imposed at the beginning of interfaces.

**constN[V]:** a N-bit integer constant of value V
**intN[min:max]:** a N-bit integer with range from min to max
**flags[(V)+, T]:** a set of constants of type T
**string**: a zero-terminated memory buffer
**array[T, min:max]:** a bounded array of elements of type T
**ptr[dir, T]:** a pointer to an object of type T; dir specifies the direction (input or output)
**len[identifier, intN]:** a N-bit integer denotes the size of another field specified by the identifier
**identifier { (identifier T)+ }:** a custom structure

**Figure 7: Supported types for syscall specifications**

**Service type identification**. One service can provide different user clients through a uniform interface `IOServiceOpen`, each of which is bound to a unique integer passed as the third argument "`type`". Hence, the driver needs to firstly check the argument to figure out which user client to instantiate. To infer the valid values, we conduct a dynamic symbolic execution on `IOServiceOpen()` with the third argument "`type`" symbolized. More specifically, SyzGen looks for class initialization (*i.e.,* `IOUserClient::IOUserClient`) of any user client during symbolic execution and then performs constraint solving against the service type to obtain the unique value. In the case where multiple values are valid, which usually means that there is only user client and thus no need for the driver to validate the service type, we simply select the minimum value. With the help of symbolic execution to explore all possible paths, SyzGen is able to discover all valid values for service type and their corresponding user client classes. Note that we terminate a path when it reaches the class initialization function and thus symbolic execution does not suffer from the notorious path explosion problem.

**Specification reduction**. Since SyzGen produces the syscall specification in the format of Syzkaller templates [26], it needs to support the data types defined in its declarative description language. Fig. 7 lists all supported types by SyzGen.

As mentioned in §4.3, SyzGen generates one specification for one explored path from symbolic execution. After path pruning by its depth, we merge two templates based on a set of simple rules defined in Table 6 in the Appendix. Basically, we observe that most specifications only differ in one field and thus can be straightforwardly merged. For example, if one specification says that one byte of the input can take a constant of 1 while another says it can take a constant of 2, we will simply merge the two specifications and say that this byte can take either the value of 1 or 2. Note that coalesced specifications can be further merged recursively until they differ by more than one field. Later we will illustrate an example template produced by SyzGen in Fig. 8.

**Fuzzing with Kernel Coverage** Coverage-guided fuzzing has become the de facto standard for fuzzing. To collect coverage for macOS kernel fuzzing, Panic [18] proposes to leverage static binary instrumentation, but it is not a full-fledged tool and not publicly available. kAFL [22] takes advantage of hardware (*i.e.,* intel-pt) and thus is agnostic to OSes. We, however, found it lacking support for the latest macOS due to the underlying virtual machine it uses (*i.e.,* qemu-pt, a customized version of qemu). Therefore, we propose a lightweight technique to collect coverage without the requirement of specific hardware, virtual machine, or source code. Basically, we leverage the built-in kernel debugger (available in all modern OSes) composed of an agent running inside the kernel to receive and execute commands and a debugger running on a remote machine to send commands to the kernel and display the results. The agent internal to the kernel is capable of setting breakpoints at specific virtual addresses by patching the code with INT3 instructions. When the breakpoint is hit, the kernel would pause and divert its execution to the agent, which in turn sends related information to the remote debugger and wait for its subsequent commands (*e.g.,* resume). By setting the breakpoints at the beginning of every basic block we are interested in, we can effectively collect the block coverage feedback. However, the communication between the in-kernel agent and a remote debugger is prohibitively expensive. Therefore, we develop another in-kernel module acting as the debugger and collect coverage natively. As an optimization, SyzGen removes breakpoints that have been hit to eliminate needless tracing overhead (as suggested by UnTracer [20]) and thus collects only block coverage. The implementation only takes 1K lines of C code and can be ported to other OSes for closed-source kernel module fuzzing since most OSes share similar designs for kernel debugging.

# 6 EVALUATION

To determine the effectiveness of SyzGen we evaluate both its interface recovery and bug-finding capabilities. Our experiments answer the following questions:

1. How is SyzGen's effectiveness on interface recovery (§6.2)?
2. How much does dependence generalization contribute (§6.3)?
3. Can SyzGen find real-world vulnerabilities (§6.4)?

## 6.1 Evaluation Setup

Since there is no prior work to generate syscall specification for macOS drivers from end to end, we evaluate SyzGen by breaking down each component. It is worth noting that we have re-implemented most related work (*i.e.,* p-joker[3] and IMF[14]) in SyzGen and even made them better. Specifically, we run the following configurations of SyzGen:

- **SyzGen-Base**. It is an improved version of p-joker with advanced symbolic execution and automated specification generation. After the step of service and command identifier determination (see §4.2), SyzGen can already produce an initial specification with the knowledge of interfaces and some simple constraints on inputs extracted from dispatch tables (whenever available). As we will show later, this configuration represents a compelling baseline, especially for those small and simple drivers.
- **SyzGen-IMF**. Though IMF [14] only works with syscall that has known specifications, its idea to infer explicit dependence from syscall logs can be applied to unknown drivers with some adaptation (see §4.1). In this configuration, we retain the explicit dependencies learned from logs but disable the dependence generalization component. Interface recovery is performed as well. This represents a strong configuration that is similar but more complete than the original IMF.

- **SyzGen**. This configuration enables all components as described in §4. Compared to SyzGen-IMF, the only difference is the signature-based dependence inference.

All experiments are conducted on three machines, a Macbook Air with 2.2 GHz Intel Core i7, a Macbook with 1.1 GHz Dual-Core Intel Core m3, and a Macbook Pro with 1.1 GHz Dual-Core Intel Core m3. For any tested driver, we ensure all related evaluations are performed on the same machine to guarantee a fair comparison. The version of tested macOS is 10.15.4, and they run in VMware Fusion 11.5.7. In total, we have tested 25 user clients as listed in Table 2. Each fuzzing campaign takes 24 hours, and we repeated it three times for each driver to report the coverage on average to reduce randomness. The only exception is the file system driver 'AppleAPFSUserClient' due to its low throughput (*i.e.,* 5 test cases per minute) caused by one time-consuming interface to create new disk volume, and thus we fuzz it for 72 hours. To collect syscall logs for drivers, we look for any macOS build-in application associated with them (*e.g.,* system preferences for Bluetooth driver) and manually perform all possible operations on it multiple times. Also, we found some sample code from Apple open source projects [1]. As a result, we successfully obtained logs for nine user clients.

## 6.2 Effectiveness of Interface Recovery

To evaluate the effectiveness of different steps of our interface recovery solution, we ran SyzGen against 254 drivers. As a result, SyzGen identified 56 valid service names in total. We found that the majority of drivers were not loaded (72%) in our environment or did not expose the interface `IOConnectCallMethod()` (18.5%). For each service name, it may correspond to multiple user clients, each of which is bound to a specific type number. SyzGen successfully discovered 60 user clients and their corresponding type numbers, among which we selectively fuzz 25 user clients as listed in Table 2. The selection of targets to fuzz is based on the code size (*i.e.,* the fifth column) and the complexity of inputs as these metrics are positively correlated to the number of bugs (see §6.4). Note that the fifth column of Table 2 shows the number of all basic blocks in a driver and does not necessarily represent the number of blocks that could be reached by the corresponding user client (there may also be blocks reachable only from handling specific hardware interrupts).

**Effectiveness of service identification**. By design, all the services registered in the system must be queryable via the API *IOServiceMatching()*. Therefore, we believe it is complete using the approach proposed in §4.2. We are unable to find any false positives or false negatives. As for the 60 user clients and corresponding type numbers that SyzGen extracts, we manually checked the binary to confirm the correctness and developed a test program to ensure those user clients were indeed reachable from userspace. However, we observed that SyzGen failed to identify the user client for one particular driver `CoreAnalyticsHub` because the user client is instantiated by some daemon after system startup, and subsequent requests for connecting are rejected unless prior instance terminates. SyzGen utilizes memory snapshots as the initial state to perform symbolic execution and thus is unable to bypass the singleton check. Additionally, judging from the class names with the suffix 'UserClient', we found two definitions of user clients (*e.g.,* `AppleUSBLegacyInterfaceUserClient`) in the binaries but cannot find any interface that can trigger the creation of them. Therefore, we do not consider them as false negatives.

**Effectiveness of command identifier determination**. The second column of Table 2 shows the number of valid command identifiers extracted from the corresponding user client. In total, SyzGen found 504 valid command identifiers across 25 user clients. Though SyzGen does not distinguish among switch cases, if-else, and function tables, we manually inspected the binaries and found that 16 user clients use dispatch function tables, eight use switch cases and one combines if-else and switch cases, demonstrating the generality of our tool. We also manually verified those extracted command identifiers were correct and SyzGen did not miss anyone. Moreover, unlike DIFUSE [9] and p-joker [28] that assume the command identifier must be passed through certain parameter (*e.g.,* the second parameter 'selector' of `IOConnectCallMethod`), SyzGen successfully recognizes the control identifier embedded in the inputStruct (*i.e.,* the fifth parameter to `IOConnectCallMethod`) for `IOBluetoothHCIUserClient`, making the subsequent steps possible.

**Effectiveness of specifications overall by coverage performance**. Since we do not have the ground truth for the syscall specifications, we compare the coverage between SyzGen and SyzGen-Base to demonstrate the overall improvement over interface models. The third and fourth columns of Table 2 shows the block coverage for SyzGen-Base and SyzGen, respectively. As we can see, the average coverage improvement is 48%, demonstrating the effectiveness of the specifications SyzGen generates. Most improvements are due to a few complex drivers where we either extrapolate many explicit dependencies (*e.g.,* 469% improvement for `IOBluetoothHCIUserClient`) or recover many constraints imposed on the inputs (*e.g.,* 306% improvement for `AppleSSEUserClient`), indicating that the coverage improvement is correlated positively with the complexity of the target. In contrast, for drivers that are small and with few input constraints to begin with (or if their constraints are already encoded in the dispatch table which can be extracted by SyzGen-Base), we see almost no improvement and even slightly worse performance (due to noise) in some cases. This is expected because such drivers may not have many interesting behaviors to test in any event. To be thorough, we investigated the missing block coverage and found that most are simply on the error paths that terminate early (which are pruned by SyzGen as described in §5), indicating that SyzGen works as expected.

In addition, we managed to find the source code of two drivers `IONetworkUserClient` and `IOAudioFamily`, allowing us to inspect the source code and confirm the quality of the corresponding templates SyzGen generated. We can confirm that SyzGen successfully recovered all the argument structures. However, SyzGen failed to identify the explicit dependency for `IOAudioFamily` due to lack of syscall logs. We also noticed a missing constraint in `IONetworkUserClient`, which requires an input string to match some pre-registered key maintained in an internal dictionary object. SyzGen fails to extract those fixed keys because it is challenging to perform symbolic execution on cryptography routines (*e.g.,* hash functions). Fortunately, the syscall logs happen to contain valid values for that string field, which are used to produce the initial corpus, mitigating the specific issue. Nonetheless, these two drivers

| User Client | #Valid Command Identifier | Block Coverage | | #Blocks |
|---|---|---|---|---|
| | | SyzGen-Base | SyzGen | |
| ACPI_SMC_PluginUserClient | 4 | 104 | 104 | 1875 |
| AppleImage4UserClient | 4 | 24 | 24 | 1645 |
| IOHDIXController | 2 | 61 | 61 | 1769 |
| AppleMCCSUserClient | 9 | **107** | 104 | 1739 |
| AppleSSEUserClient | 1 | 62 | **252** | 830 |
| AppleCredentialManagerUserClient | 2 | 224 | **758** | 8043 |
| AHCISMARTUserClient | 9 | 282 | **302** | 2766 |
| AppleFDEKeyStoreUserClient | 27 | 108 | **178** | 824 |
| IOAudioEngineUserClient | 6 | 504 | 504 | 3875 |
| AppleAPFSUserClient | 49 | 6232 | **6811** | 37889 |
| IOBluetoothHCIUserClient | 213 | 1014 | **5773** | 17989 |
| IOAVBNubUserClient | 21 | **453** | 452 | 1266 |
| IONetworkUserClient | 5 | 157 | 157 | 3806 |
| IOReportUserClient | 4 | **133** | 132 | 263 |
| IOHDACodecDeviceUserClient | 2 | 120 | **123** | 519 |
| AppleHDAControllerUserClient | 2 | 217 | **223** | 3069 |
| AppleHDADriverUserClient | 2 | 905 | **1032** | 24920 |
| AppleHDAEngineUserClient | 10 | 1367 | 1367 | 24920 |
| AppleUpstreamUserClient | 7 | 183 | **241** | 492 |
| AppleUSBHostInterface | 34 | **1903** | 1719 | 15408 |
| AppleUSBHostFrameworkInterface | 26 | **2949** | 2925 | 15408 |
| AppleUSBHostDeviceUserClient | 20 | 3220 | **3255** | 15408 |
| AppleUSBHostFrameworkDevice | 13 | 1373 | 1373 | 15408 |
| AppleUSBLegacyDeviceUserClient | 25 | 255 | 255 | 14019 |
| AudioAUUCDriver | 7 | 158 | **255** | 439 |
| Overall | 504 | | | |

**Table 2: Tested macOS drivers**

| User Client | #Dependencies* | | Block Coverage | |
|---|---|---|---|---|
| | SyzGen-IMF | SyzGen | SyzGen-IMF | SyzGen |
| AudioAUUCDriver | 2 | 5 | 209 | 255 |
| AppleAPFSUserClient | 4 | 21 | 6503 | 6811 |
| AppleUpstreamUserClient | 2 | 5 | 192 | 241 |
| IOBluetoothHCIUserClient | 23 | 235 | 4421 | 5773 |
| IONetworkUserClient | 2 | 5 | 157 | 157 |
| Overall | 33 | 271 | 11482 | 13237 |

*: The number of dependencies is counted by the usage of them.

**Table 3: Comparison of dependence inference between SyzGen-IMF and SyzGen**

are rather simple (*i.e.,* most interfaces only require an integer and a string) and may not be representative.

We further reverse engineered all tested drivers to obtain some ground truth with our best effort. In general, we believe the automatically-generated specifications were not entirely precise but good enough. For instance, it is sufficient for a boolean field of size 8 bytes to have two values (*i.e.,* True and False), but our specifications may specify a valid range of [0, MAX_INT], causing the fuzzer to mutate the value unnecessarily (likely end up with the same True value). One weakness we find is the inability to express complex relationships between fields of structures in the specification due to the limitation of description language defined by Syzkaller. For example, Syzkaller template cannot express a relationship such as "field A should always be twice the value of field B". This prevents us from exploring certain interesting code paths in the driver. Additionally, for some drivers (*e.g.,* `IOAVBFamily`), we find that they allow users to provide some string as a key to create an object (*e.g.,* `addAVBClient(char* key)`) and later on input the same key to delete the corresponding object (*e.g.,* `removeAVBClient(char* key)`). Currently, SyzGen fails to recognize such string-based dependence variables and the corresponding dependencies.

## 6.3 Dependence Generalization

To see how much benefit does dependence generalization provides, we compared SyzGen against SyzGen-IMF in terms of the number of identified dependencies and block coverage. As shown in Table 3, SyzGen-IMF can infer dependencies for 5 user clients among those with logs. Interestingly, starting from 33 dependencies learned from logs by SyzGen-IMF, SyzGen can generalize them to those interfaces without logs and recognize 238 more dependencies. Note that the number of dependencies is counted by the instances of them among different interfaces. For example, SyzGen-IMF only infers three types of explicit dependencies (identified by the dependence variable from the *generate* interface) from the logs for `IOBluetoothHCIController`, in which one represents the request the application sends to the service, one represents a remote Bluetooth device, and one represents the connection between two devices. From the logs, SyzGen-IMF discovered 16, 2, and 5 *use* interfaces that take the three types of dependence variables, respectively. The fourth and fifth columns of Table 3 further demonstrate the coverage improvement achieved by those additionally discovered dependencies. On average, SyzGen achieves 16.5% more coverage compared to SyzGen-IMF. Note that we only compare SyzGen against SyzGen-IMF for 5 user clients because we did not even find any explicit dependence for the rest 20 user clients. This means that SyzGen and SyzGen-IMF degrades to the same mode where only interface recovery is performed. Upon a closer look, these user clients mostly correspond to simple and small drivers.

To get a better intuition on the evolution of the fuzzing process, we visualize the block coverage over time as shown in Fig. 9 in the Appendix. The coverage improvement in general is as significant as we expected. We investigated the reasons and come to the following conclusions. Since we uncovered more explicit dependencies, the search space of fuzzing for input is enlarged. Thus, in general more fuzzing time is needed to cover more basic blocks. For example, we can see from Fig. 9(b) that the coverage clearly still improves towards the end of the experiment for SyzGen. However, there are exceptions. For `IOBluetoothHCIUserClient`, we observe only 30.7% coverage improvement over SyzGen-IMF, and yet SyzGen identified 10 times more explicit dependencies as shown in Table 3. It turns out that `IOBluetoothHCIUserClient` serves mostly as a middleware connecting userspace applications and the underlying firmware. Thus, most interfaces simply construct the request from user inputs and forward it to the firmware through a common set of functions, leaving much smaller space for coverage improvement. Nevertheless, we are able to find serious vulnerabilities in the new interfaces as will be shown in §6.4. For `IONetworkUserClient` and `AppleAPFSUserClient`, the improvement is relatively small due to their unique characteristics of explicit dependence. As opposed to most drivers in which the value of a dependence variable is dynamically allocated (*e.g.,* object ID), these two drivers in fact have some pre-defined IDs that can be directly used. For instance, interface `methodVolumeSpace()` in driver `AppleAPFSUserClient` provides the space information of a given volume represented by a dependence variable which can be either produced by the interface

| User Client | #Interfaces | #Interfaces w/ Traces |
|---|---|---|
| AppleAPFSUserClient | 49 | 5 |
| AppleCredentialManagerUserClient | 2 | 1 |
| AppleSSEUserClient | 1 | 1 |
| AppleUpstreamUserClient | 7 | 2 |
| AppleUSBHostDeviceUserClient | 20 | 4 |
| AppleUSBHostInterfaceUserClient | 34 | 3 |
| AudioAUUCDriver | 7 | 2 |
| IOBluetoothHCIUserClient | 213 | 16 |
| IONetworkUserClient | 5 | 2 |
| Overall | 338 | 36 |

**Table 4: Numbers of interfaces with traces.**

`methodVolumeCreate()` or some special constants such as zero representing the default volume.

**Dependence verification**. Although we do not have the ground truth for the dependencies (except IONetworkUserClient and IOAudioFamily manually verified as aforementioned), we manually reverse engineer the binaries to confirm the correctness of them (*i.e.,* no false positives). Besides, we find some hints from those meaningful functions' names, indicating the presence of dependencies. We take the driver APFS as an example, in which functions `methodVolumeDelete` obviously requires a dependence value produced by the function `methodVolumeCreate`. We also observed missing explicit dependence in 4 user clients where we have no trace at all and thus cannot infer any explicit dependency.

**Necessity for explicit dependence generalization**. IMF [14] targets on commonly used syscalls and thus downloads 5 most popular and free apps in each category from Apple App Store to collect syscall logs, while Moonshine [21] focuses on Linux core subsystem and relies on existing test suite such as Linux Testing Project (LTP), Linux Kernel selftests (kselftests), Open Posix Tests, and Glibc Testsuite. In contrast, drivers oftentimes lack test suites and applications exercising every interface. In our evaluation, we successfully obtained traces for 9 services listed in Table 4. In total, only 36 out of 338 interfaces have traces, resulting in 238 more dependencies being neglected initially by SyzGen-IMF as shown in Table 3. Given the scarcity of traces, generalizing explicit dependencies from interfaces with traces to those without traces could reduce the reliance on collecting traces and improve interface recovery.

## 6.4 Bug Finding and Case Studies

During the evaluation of fuzzing (§6.2), we collected thousands of crash logs[3] and crashing test cases, manually triaged them, and filtered out duplicates based on the stack trace. In total, SyzGen was able to find 34 unique bugs in 25 user clients. Table 5 lists all the bugs SyzGen found as well as their corresponding services and crashing types including arbitrary read, OOB read, integer overflow, null pointer deference, etc. In general, as expected, we find that the number of bugs is positively correlated to the code size and complexity of the inputs.

The base configuration of SyzGen (with service and command identifier determination and constraints recovered from dispatch table if applicable) is able to find 29 bugs. Although the result is impressive on its own, we found that all of them are rather shallow bugs that can be easily triggered, *e.g.,* invalid userspace pointer could trigger an assertion failure. In contrast, those bugs only identified by SyzGen require either complex inputs or correct handling of dependencies. They also have more serious security impacts. One of which can even lead to privilege escalation.

We are currently working on responsibly disclosing those vulnerabilities to Apple. So far, we have received 2 CVE numbers. In the rest of this subsection, we will present the case studies of several bugs, explaining their root causes and demonstrating how SyzGen is able to discover them.

•**Incoherent checks**. One of the most interesting bugs in our collection is caused by incoherent checks against a 4-byte boolean input. Depending on its value, the driver expects different sizes of inputs. The problem is that the driver initially sanitizes the inputs by checking the least significant byte of the boolean input, but considers the whole 4 bytes as a boolean value when consuming the rest input. Due to the incoherent checks, a deliberately crafted boolean value (*e.g.,* 0x100) could cause different outcomes, leading to an out-of-bound read. This subtle difference can be easily neglected by manual audits. Thanks to symbolic execution to explore every possible path, SyzGen is able to model different paths in different specifications, including one modeling the boolean value of zero, one with the value larger than zero, and one requiring only the least significant byte to be zero.

•**Nested structure with dependencies and inter-fields relationship**. Fig. 8 showcases the specification for the interface

| User Client | Vuln Type | Status | Found* |
|---|---|---|---|
| IOBluetoothHCIUserClient | Arbitrary Read | CVE-2020-9929 | A |
| | Arbitrary Read | Confirmed | A |
| | Null Pointer | Confirmed | A |
| | OOB Read | Reported | A |
| | OOB Read&Write | CVE-2020-9928 | A |
| ACPI_SMC_PluginUserClient | Null Pointer | Reported | A&B |
| | Null Pointer | Reported | A&B |
| IOHDIXControllerUserClient | Out of Memory | Reported | A&B |
| AppleCredentialManagerUserClient | Invalid Free | Reported | A |
| AppleAPFSUserClient | Memory Leak | Reported | A&B |
| | Assert Failure | Reported | A&B |
| AppleUSBLegacyDeviceUserClient | Assert Failure | Reported | A&B |
| AppleUSBHostInterfaceUserClient | Null Pointer | Reported | A&B |
| | Null Pointer | Reported | A&B |
| | Integer Overflow | Fixed | A&B |
| | Assert Failure | Reported | A&B |
| | Assert Failure | Reported | A&B |
| | Assert Failure | Reported | A&B |
| AppleUSBHostFramework-DeviceClient | Null Pointer | Reported | A&B |
| | Null Pointer | Reported | A&B |
| | Assert Failure | Reported | A&B |
| AppleUSBHostDeviceUserClient | Kernel Hang | Reported | A&B |
| | Kernel Hang | Reported | A&B |
| | Assert Failure | Reported | A&B |
| | Assert Failure | Reported | A&B |
| AppleUSBHostFramework-InterfaceClient | Integer Overflow | Fixed | A&B |
| | Assert Failure | Reported | A&B |
| IOHDACodecDeviceUserClient | Null Pointer | Reported | A&B |
| | Null Pointer | Reported | A&B |
| | Kernel Hang | Reported | A&B |
| AppleHDAControllerUserClient | Null Pointer | Reported | A&B |
| | Null Pointer | Reported | A&B |
| AppleHDADriverUserClient | Null Pointer | Reported | A&B |
| | Null Pointer | Reported | A&B |

*: A: SyzGen; B: SyzGen-Base

**Table 5: Vulnerabilities found by SyzGen**

[3]MacOS would automatically generate crash reports containing backtraces upon panics.

```
resource port[io_connect_t]
resource connection_0[int32]
resource connection_1[int16]
Group199_3_struct_48 {
    … …
    Group199_3_buffer_11  int64
    … …
} [packed]
Group199_3_struct_46 {
    Group199_3_ptr_6  ptr[in, connection_0]
    Group199_3_ptr_8  ptr[in, connection_1]
    Group199_3_ptr_35  ptr[in, Group199_3_struct_48]
    Group199_3_buffer_36  array[const[0, int8], 32]
    Group199_3_const_37  len[Group199_3_ptr_6, int64]
    Group199_3_const_38  len[Group199_3_ptr_8, int64]
    Group199_3_const_39  len[Group199_3_ptr_35, int64]
    Group199_3_buffer_40  array[const[0, int8], 32]
    Group199_3_const_41  const[199, int32]
} [packed]
syz_IOConnectCallMethod$Group199_3(connection port, selector
const[0], input ptr[in, const[0, int8]], inputCnt const[0], inputStruct ptr[in,
Group199_3_struct_46], inputStructCnt const[116], output ptr[out,
const[0, int8]], outputCnt ptr[in, const[0, int32]], outputStruct ptr[out,
const[0, int8]], outputStructCnt ptr[in, const[0, int32]])
```

**Figure 8: Syscall specification where resource is the keyword for dependencies.**

that could trigger an arbitrary read bug in the Bluetooth driver, which has been assigned CVE-2020-9929. In this example, resources 'connection_0' and 'connection_1' are two types of dependencies that are inferred through our proposed signature-based dependence inference approach. As we can see, the fifth argument (*i.e.,* inputStruct) is a nested structure that consists of multiple fields of different types, including pointer, array, constant, and so on. Additionally, the specification specifies inter-fields relationship, *e.g.,* the length field 'Group199_3_const39' represents the size of another structure 'Group199_3_struct_48'. The vulnerability results from a memory read whose address is directly provided by user (*i.e.,* Group199_3_buffer11) without any sanitization. That said, to trigger the bug, we must properly construct the input and set up the correct sequence of syscalls to obtain valid dependence values for `connection_0` and `connection_1`. Thus, without dependence inference and interface recovery, it would be difficult for fuzzing to properly instantiate the arguments to the syscall, likely missing this bug.

**Design issue in Bluetooth**. A common practice for macOS drivers to deal with race conditions is to enforce a single-threaded work loop which ensures sequential execution of requests. However, the problem with this design is that some requests need to communicate with the underlying firmware which in turn may communicate with other devices, and thus occupying the working thread while waiting for the response can block the entire execution and is not desired. To cope with it, the driver must put the awaiting thread to sleep until any response arrives, which unfortunately leaves a loophole for race conditions. For a waiting request that has not been completed, any associated global data are susceptible to the modification of following requests. We found that this issue is prevalent in the Bluetooth driver and cannot be fixed without substantial changes to the design of the system. This vulnerability can be exploited to achieve privilege escalation and has been assigned CVE-2020-9928.

## 7 DISCUSSION AND LIMITATION

Even though we have shown SyzGen as a promising direction to generate templates for closed-source kernel modules, there are still improvements that can make the solution even better. One premise of fuzzing and any dynamic analysis is that the target driver must be loaded so that we could invoke its interfaces. However, we find that the majority of drivers are not running on our tested machines. Nonetheless, it is arguable that only those loaded-by-default drivers are more meaningful attack surfaces. Also, since SyzGen begins with logs to infer explicit dependencies and then generalizes them beyond the logs, it would degrade to the mode where only interface recovery is performed if no log is available.

Modern fuzzing is typically not only coverage-guided but also usually accompanied by various sanitizers (*e.g.,* Kernel Address Sanitizer or KASAN) that could catch various types of bugs even when they do not cause an immediate kernel crash. Unfortunately, retrofitting sanitizers into closed-source binaries (especially kernel drivers) remains to be a challenge. Static rewriting of binaries is a possible direction to address this problem but at the moment only ELFs binaries can be rewritten with a high accuracy [12]. QASan [13] is an alternative that utilizes QEMU to dynamically instrument the binary, though it only supports user-mode programs. Apple occasionally releases a few driver binaries with KASAN enabled, but we found that only three drivers we tested had this feature. Windows is equipped with an in-house driver verifier to monitor drivers by manipulating memory allocation and resource management, which can be integrated into our system if we port SyzGen to Windows.

## 8 CONCLUSION

In this paper, we proposed SyzGen, a first attempt to automatically generate specifications to fuzz drivers without source code. SyzGen could infer explicit dependencies for interfaces by analyzing a small number of execution traces collected from exiting applications, and then generalize the knowledge to other interfaces without traces. Instead of producing syscall specifications in one shot, SyzGen yields coarse-grained specifications at the beginning and iteratively refines them, allowing us to combine knowledge learned from multiples runs under different calling contexts. We also proposed a lightweight coverage collection technique to guide fuzzing without requiring any specific hardware, virtual machine or kernel source code. Our empirical evaluation shows that SyzGen is effective in recovering driver interfaces, including input structure, constraints upon inputs, and explicit dependencies between syscalls. Our evaluation shows that SyzGen is effective in producing high-quality syscall specifications, leading to 34 unique bugs, including one that attackers can exploit to escalate privilege, and 2 CVEs to date.

## 9 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for providing insightful feedback on our work. This work was supported by the National Science Foundation under Grant No. 1652954.

## REFERENCES


[1] 2021. Apple Open Source. https://opensource.apple.com/source.
[2] 2021. IDA Pro. https://www.hex-rays.com/ida-pro/.
[3] 2021. p-joker. https://github.com/lilang-wu/p-joker.
[4] 2021. Symbolic lengths. https://docs.angr.io/advanced-topics/gotchas.
[5] 2021. Syzbot. https://syzkaller.appspot.com/upstream.
[6] 2021. Syzkaller. https://github.com/google/syzkaller.
[7] Xiaolong Bai, Luyi Xing, Min Zheng, and Fuping Qu. 2020. iDEA: Static Analysis on the Security of Apple Kernel Drivers. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1185–1202.
[8] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices* 46, 3 (2011), 265–278.
[9] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.
[10] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. 2008. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*. 391–402.
[11] William HE Day and Herbert Edelsbrunner. 1984. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of classification* 1, 1 (1984), 7–24.
[12] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 151–163.
[13] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. 2020. Fuzzing binaries for memory safety errors with QASan. In *2020 IEEE Secure Development (SecDev)*. IEEE, 23–30.
[14] HyungSeok Han and Sang Kil Cha. 2017. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2345–2358.
[15] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.
[16] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid fuzzing on the linux kernel. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*.
[17] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. (2011).
[18] Juwei Lin and Junzhi Lu. 2019. Panic on the Streets of Amsterdam: PanicXNU 3.0. 2019 HITB Security Conference.
[19] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*. 1–1.
[20] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 787–802.
[21] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. 729–743.
[22] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kafl: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*. 167–182.
[23] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*.
[24] JV Stoep. 2016. Android: protecting the kernel. *Linux Securit Summit (August 2016)* (2016).
[25] Dmitry Vyukov. 2019. Syzkaller: an unsupervised, coverage-guided kernel fuzzer.
[26] Dmitry Vyukov. 2020. Syzkaller: adventures in continuous coverage-guided kernel fuzzing. Bluehat IL.
[27] Fish Wang and Yan Shoshitaishvili. 2017. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 8–9.
[28] Lilang Wu and moony Li. 2019. Fresh Apples: Researching new attack interfaces on iOS and OSX. In *HITB Security Conference*.
[29] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1643–1660.
[30] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and scalable detection of double-fetch bugs in OS kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 661–678.
[31] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 818–834.
[32] Michal Zalewski. 2015. American fuzzy lop. *URL http://lcamtuf. coredump. cx/afl* (2015).


**Algorithm 1:** Locate command identifier and collect its valid values

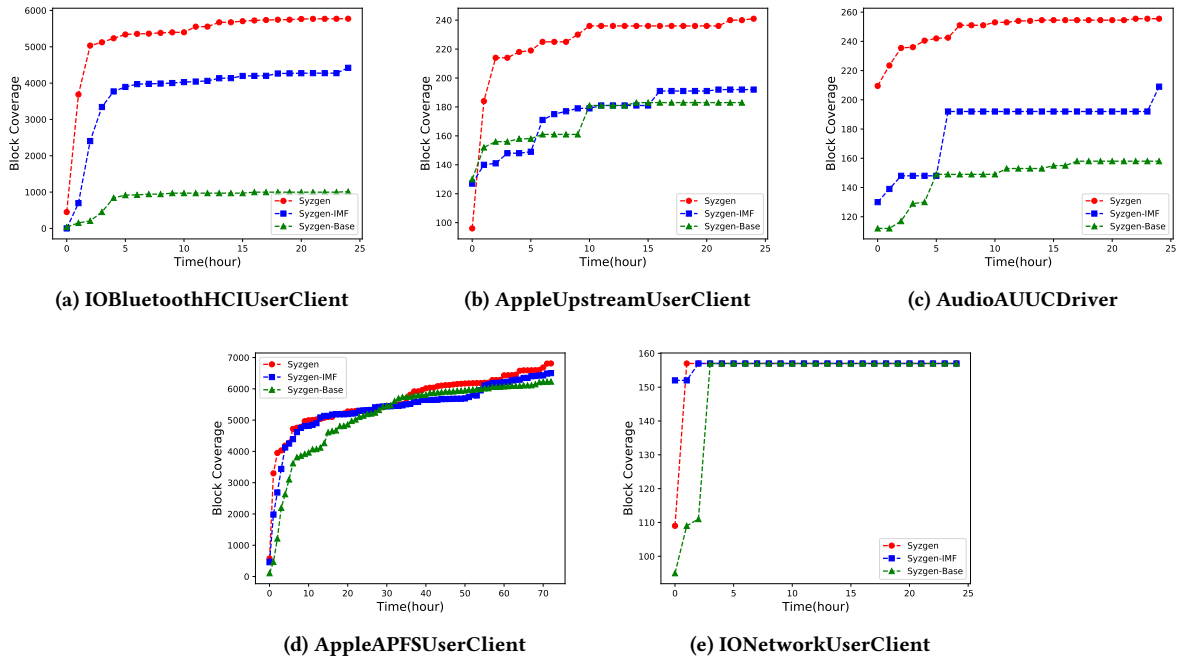
```
Function AnalyzeCtrlID(α:init state, τ : all class member functions):
    Symbolize all inputs for α
    actives, deferred ← [α], []
    while actives is not empty do
        actives ← SymbolicExecution(actives)        ▷ Step forward all states by one basic block
        foreach s in actives do
            if s.addr in τ then
                move s from actives to deferred
        if actives is empty then
            if all states in deferred have the same address then
                swap(actives, deferred)
                continue
            cmds ←find common symbolic variables from states in deferred
            foreach cmd in cmds do
                if cmd can have different values in different states from deferred then
                    foreach s in deferred do
                        if cmd can have multiples values in s then
                            move s from deferred to actives
                    if actives is empty then
                        return cmd, values for cmd
                    break
            if actives is empty then
                Randomly move one state from deferred to actives
```

## A COMMAND IDENTIFIER DETERMINATION

Algorithm 1 describes the procedure to identify the command identifier, as well as its valid values and corresponding functionalities (*i.e.,* function address). We observed that the entry function `IOConnectCallMethod` is simply a dispatch function that calls other functions depending on the command identifier. Based on which, `SyzGen` considers all functions inside the target driver as candidates for functionalities, and performs symbolic execution to locate the key variable for dispatching. Essentially, `SyzGen` symbolizes the inputs and employs a breadth-first search strategy to explore all paths, during which it suspends any state that runs into a function that could potentially be the entry of one functionality. If no states are active, `SyzGen` extracts common symbolic variables from constraint sets of those stopped states and check whether

| Type | const[j] | flags[$j_0$, ..., $j_m$] | ptr | int[$j_{min}$, $j_{max}$] | struct B |
|---|---|---|---|---|---|
| const[i] | flags[i, j] | flags[i, $j_0$, ..., $j_m$] | ptr if i == 0 | int[$j_{min}$, $j_{max}$] if $j_{min}$ <= i <= $j_{max}$<br>int[$j_{min}$-1, $j_{max}$] if $j_{min}$-1 == i<br>int[$j_{min}$, $j_{max}$+1] if $j_{max}$+1 == i | B if B.fields[0] == const[i] |
| flags[$i_0$, ..., $i_n$] | | flags[$i_0$, .. $i_n$, $j_0$, ... $j_m$] | | int[min, max] if $\forall x \in [min, max], x \in [j_{min}, j_{max}] \cup [i_0, ..., i_n]$ | B if B.fields[0] == flags[$i_0$, ..., $i_n$] |
| ptr | | | | | B if B.fields[0] == ptr |
| int[$i_{min}$, $i_{max}$] | | | | int[min, max] if $\forall x \in [min, max], x \in [j_{min}, j_{max}] \cup [i_{min}, i_{max}]$ | B if B.fields[0] == int[$i_{min}$, $i_{max}$] |
| struct A | | | | | B if $A \subset B$ |

**Table 6: Rules for merging two specifications if they only differ in one field. We simply take the union of two fields whenever possible.**




**(a) IOBluetoothHCIUserClient** **(b) AppleUpstreamUserClient** **(c) AudioAUUCDriver**

**(d) AppleAPFSUserClient** **(e) IONetworkUserClient**

**Figure 9: Coverage for SyzGen-IMF, SyzGen-IMF and SyzGen.**

there is one that could have unique values in different states. If so, SyzGen believes that symbolic variable is the `command identifier`, and it not only tells which parameter `command identifier` comes from (*e.g.*, inputStruct which is the fifth parameter to "IOConnectCallMethod") but also precisely locates where it is (*e.g.*, first 8 bytes of inputStruct) since it can be embedded in a nested structure. Otherwise, SyzGen resumes some states and repeats the process until all states terminate.