Demystifying the Dependency Challenge in Kernel Fuzzing

Yu Hao
UC Riverside
Riverside, USA
yhao016@ucr.edu

Hang Zhang
Georgia Institute of Technology
Atlanta, USA
hzhan033@ucr.edu

Guoren Li
UC Riverside
Riverside, USA
gli076@ucr.edu

Xingyun Du
UC Riverside
Riverside, USA
dxing003@ucr.edu

Zhiyun Qian
UC Riverside
Riverside, USA
zhiyunq@cs.ucr.edu

Ardalan Amiri Sani
UC Irvine
Irvine, USA
ardalan@uci.edu

ABSTRACT

Fuzz testing operating system kernels remains a daunting task to date. One known challenge is that much of the kernel code is locked under specific kernel states and current kernel fuzzers are not effective in exploring such an enormous state space. We refer to this problem as the dependency challenge. Though there are some efforts trying to address the dependency challenge, the prevalence and categorization of dependencies have never been studied. Most prior work simply attempted to recover dependencies opportunistically whenever they are relatively easy to recognize. In this paper, we undertake a substantial measurement study to systematically understand the real challenge behind dependencies. To our surprise, we show that even for well-fuzzed kernel modules, unresolved dependencies still account for 59% - 88% of the uncovered branches. Furthermore, we show that the dependency challenge is only a symptom rather than the root cause of failing to achieve more coverage. By distilling and summarizing our findings, we believe the research provides valuable guidance to future research in kernel fuzzing. Finally, we propose a number of novel research directions directly based on the insights gained from the measurement study.

ACM Reference Format:
Yu Hao, Hang Zhang, Guoren Li, Xingyun Du, Zhiyun Qian, and Ardalan Amiri Sani. 2022. Demystifying the Dependency Challenge in Kernel Fuzzing. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510126

1 INTRODUCTION

Fuzzing has become one of the most popular and essential methods for uncovering bugs and vulnerabilities due to its practicability and effectiveness. Both academia and industry have devoted a large amount of resources to researching and deploying fuzzing systems. For example, Google allocated more than 25,000 servers to deploy its ClusterFuzz infrastructure and has found tens of thousands of vulnerabilities so far [13].

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510126

However, despite the continuous research efforts and improvements on fuzzing techniques, fuzzing sophisticated stateful software (*e.g.,* OS kernels) remains a big challenge. Even with the state-of-the-art kernel fuzzer Syzkaller [15], the achievable code coverage of various kernel modules is generally low from our measurements (from 21% to 46%) after extensive fuzzing sessions.

One common belief of the major obstacle is that a large portion of code can only be covered under specific kernel states (*e.g.,* some global variables need to be set to specific values). It is known to be difficult for a fuzzer to search the enormous state space and enter the desired ones. This problem cannot be easily solved by simply adopting advanced mutation strategies that operate solely on local inputs (*e.g.,* function arguments), since many conditions in the code are controlled by global memory (which may need to be set by a completely different function/syscall invocation). In this paper, we refer to this challenge as the *dependency challenge* in fuzzing OS kernels.

In recent years, there have been a few related works that made some progress in tackling this challenge. MoonShine [25] aims to recognize dependencies from manually curated test cases (*e.g.,* Linux testing project [10]), and distill them into more concise seeds for fuzzing. However, its goal is not to discover and analyze new dependencies. HFL [19] applies static analysis and symbolic execution to solve a subset of dependency issues. Although it manages to achieve improvements on code coverage compared to the baseline, according to our measurement, its result is still far from satisfactory (*e.g.,* only 10.5% of Linux kernel code can be covered during a 50-hour fuzzing session). Furthermore, it still remains unknown what types of dependencies are resolved and how large of a fraction they represent. In general, due to the lack of a deep and quantitative understanding of the dependency challenge, we argue that current research has only explored the surface of this challenge.

In this paper, we take a data-driven approach to systematically understand the dependency challenge in kernel fuzzing, with an in-depth measurement study on a subset of Linux kernel modules. Specifically, we aim to answer the following questions:

(1) How well does the state-of-the-art fuzzer perform (*e.g.,* regarding code coverage) against the complex and stateful kernel? How many uncovered branches are related to dependencies? We specifically focus on well-fuzzed kernel modules as the results will represent the difficult cases that are worth solving in the future.

(2) What are the root causes behind unresolved dependencies and how important are they? Though the community has been

aware of the dependency for years, it is still unclear what the root causes are for fuzzers' failure to resolve the dependency.

(3) What can we do to address unresolved dependencies? With the understanding of root causes, we hope to shed light on potential solutions and draw attention to blind spots in current research directions.

To answer these questions, we develop a pipeline of measurement infrastructure, including a combination of automated components (fuzzing and static analysis) and meticulous manual investigations whenever necessary to get to the root causes that are otherwise difficult to obtain. Through our measurement study, we not only discover that the dependency challenge is ubiquitous, but also find a diverse set of root causes, which need focused efforts moving forward. For example, as will be discussed in §7, we suggest that multi-syscall analysis and multi-interface fuzzing are two promising directions. Furthermore, we show that much of the kernel code is simply impossible to be covered (dead code) without a proper environment. We believe the curated dataset from our study is valuable in guiding future research in kernel fuzzing.

We summarize our main contributions as below:

- To the best of our knowledge, we are the first to perform an in-depth measurement and systematic analysis on the dependency challenge in kernel fuzzing.
- We develop a measurement pipeline including automated components that can quantify the magnitude of the dependency challenge. We will open source the measurement pipeline and data to facilitate future research.
- Based on our measurement results and deep understanding of the dependency challenge, we reveal many unexpected observations and point to new research directions that will improve kernel fuzzing.

2 BACKGROUND

2.1 Definition

This section gives some definitions to help explain the challenges of resolving dependencies and the root causes of uncovered code.

Unresolved condition (UC). We define an unresolved condition to be a condition that has been evaluated during fuzzing, but it has never been evaluated to the desired value that allows an uncovered branch to be taken.

Global memory. We define global memory to include global variables and any heap memory reachable from global variables (*e.g.,* a global pointer may point to a heap object). Either way, global memory persists across system call invocations, *i.e.,* system calls can change the global memory, which becomes observable to another.

Kernel state. We define the kernel state as all the content in global memory. If a function reads the global memory in a condition, the corresponding kernel state could influence which branch is taken. This means that an execution path of a system call depends on not only arguments but also the kernel state.

Unresolved dependency (UD). We define an unresolved dependency to be an UC where some global memory is read, *i.e.,* this condition can only be satisfied if the kernel has a desired state.

Write statement (WS). We define a statement as a *write statement* as long as it writes to some global memory.

```
1  static int cdrom_read_cdda(...) {
2    if (cdi->cdda_method==CDDA_BPC_FULL && nframes>1){
3      // uncovered code
4      ... } }
5  int register_cdrom(...) {
6    cdi->cdda_method = CDDA_BPC_FULL; }
```

Figure 1: An example case of unresolved dependency

Effective write statement (EWS). We define an effective write statement to be a WS that has the potential to change the global memory to an expected value by an UD.

Dependency. We define a dependency to be a relationship between an UD and its EWS. Conceptually it is similar to def-use, except dependency is specific to global memory and the use has to be a condition check. Furthermore, we further categorize dependencies into **explicit dependencies** and **implicit dependencies**. The former represents the cases where the value written to the global memory (*e.g.,* a file descriptor) is propagated back to userspace, which is then subsequently passed as an argument that is checked against the global memory (a read). As an example, when a syscall generates a kernel file object (through `open()`), it will be stored in a global array and its file descriptor returned to userspace. The same descriptor value will be passed in a subsequent syscall (`read()`) which causes the stored object in the array to be read. The latter represents the cases where there is no overlap between one syscall's output and another syscall's input arguments. The example we will show next is an implicit dependency.

2.2 An Example Case

We use a simple example from the *cdrom* kernel module to illustrate how to find the root cause of an UD. As shown in Figure 1, condition `cdi->cdda_method == CDDA_BPC_FULL` (line 2, where `cdi` is a global structure) always evaluates to false when a fuzzing test case reaches it, causing the true branch to be uncovered.

A common way to resolve such a condition is to find and execute its EWS [35]. In this example, line 6, which is reachable through another function `register_cdrom()`, is an EWS for the UD at line 2 since it writes the expected value to `cdi->cdda_method`. The next step is to assemble a test case reaching (and thus executing) the EWS before the UD. If the EWS is again guarded behind another UD, we need to resolve it at first - this can be a recursive process.

3 MOTIVATION

Linux Kernel Fuzzing Tackling Dependencies. The typical interface exposed by OS kernels is syscalls. Therefore, before one can start fuzzing the kernel, it is necessary to generate a test case consisting of a sequence of syscalls (with corresponding arguments prepared). In the case of the state-of-the-art kernel fuzzer Syzkaller, it requires a description or specification of the syscall interfaces. Typically the specifications (also called *templates*) are manually curated, which includes the information about the syscalls, their explicit dependency relationship, and the possible range of values of syscall arguments. Since manually curating templates for various kernel modules is not scalable especially a long tail of device drivers, there has been some recent work, *e.g.,* DIFUZE [7] and SyzGen [4], that aims to automate the generation of syscall *templates*, which

includes some limited hardcoded knowledge about explicit dependencies as well. Another direction considered is to simply take syscall traces generated by existing applications (that exercise a specific target module) and mutate the trace [16]. This bypasses the need to generate syscall templates and instead pushes the problem to being able to generate high quality syscall traces. In practice, it often falls short in coverage because existing applications typically exercise only a small portion of the kernel module.

Furthermore, Syzkaller templates by design are unable to encode implicit dependencies where no explicit return and argument relationship exists, *e.g.*, the example in Figure 1 is such a case. It is possible to analyze existing syscall traces generated by applications [16, 25]. This line of work can capture dependencies (both explicit and implicit) that are naturally exercised by existing applications, but will not be able to resolve new dependencies. Another direction (*e.g.*, explored by HFL [19]) is to conduct more sophisticated program analyses and look for read/write relationships of the same global memory. However, it focuses on resolving mostly explicit dependencies which are typically encoded in well-written templates already, i.e., the templates correctly describe most of the system calls and their relationships. Indeed, as we find out from measuring the dependency challenge in popular kernel modules, the majority of UDs are implicit instead of explicit.

Motivation. To summarize, we believe the state-of-the-art kernel fuzzing research has only explored the surface of the dependency challenge. ***In particular, we find that most of the existing work focuses on the low-hanging fruits of improving fuzzing coverage against the kernel modules with largely incomplete (or missing) syscall templates.*** For example, DIFUZE [7] generates templates (considering some explicit dependencies) for kernel modules where no prior templates exist. HFL [19] focused on discovering explicit dependencies in less tested modules. ***However, in this paper, we aim to understand what the remaining challenges are when syscall templates are already comprehensive with the obvious dependencies recognized and encoded, and when fuzzing time is sufficiently long. In particular, we are interested in whether the dependency challenge is still the prominent hurdle and why?*** Consider the example case in Figure 1. Surprisingly, even such a seemingly trivial dependency can be extremely challenging for the state-of-the-art kernel fuzzer to resolve and cannot be solved by prior work (because it is an implicit dependency). Interestingly, as will be shown in §6.3, it turns out the root cause is that the EWS is inside the initialization functions of the kernel module, and a kernel fuzzer implicitly assumes such functions are out of the fuzzing scope and never considers fuzzing such functions.

Following this direction, in this project, we select a set of kernel modules whose templates are well-written so that we can observe and analyze their coverage deficiencies. From this study, the obvious question we hope to answer is whether dependencies are still a major hurdle. If so, what future research directions should be considered to overcome the challenge? To achieve this goal, we rely on the state-of-the-art kernel fuzzer, Syzkaller, which has proved effective in finding thousands of kernel bugs [14] and continuously improved. In particular, we allow Syzkaller to fuzz specific kernel modules sufficiently long to realize the "full" potential of the corresponding templates. This allows us to have an objective view of the remaining UCs, which represent the hard cases we want to further investigate.

4 MEASUREMENT PIPELINE

To conduct a proper analysis of the dependency challenge in kernel fuzzing, we build a measurement pipeline that has a combination of automatic and manual analysis components. Roughly speaking, the automatic analysis aims to measure the scale of the dependency problem, whereas the manual analysis attempts to distill the root causes of UDs. The latter is often much more in-depth and requires more than just mechanical analysis.

The high-level workflow of the measurement pipeline is shown in Figure 2. First, after long fuzzing sessions of specific kernel modules, we collect the coverage and test cases from a fuzzer and obtain the UCs based on the coverage. Then we use static analysis to determine what UCs are actually UDs, which can quantify the degree of the dependency challenge (answering the first question in §1). In addition, we leverage static analysis to recognize the corresponding WSs that can influence a condition, which is a necessary step toward analyzing the root causes of UDs (in part to answer the second question in §1). From here on, we manually inspect each WS of a dependency and figure out EWS. If so, we then investigate the reason Syzkaller fails to resolve the dependency. If none of the WSs can resolve the dependency, we will also investigate the underlying reasons.

Step 1: Collecting Coverage. This step is relatively straightforward. In addition to allocating sufficient fuzzing time for kernel modules, we log all the test cases and their corresponding coverage such that we can later use them for automated and manual analysis.

Step 2: Determining Unresolved Dependencies. The next step is to determine what UCs are UDs. This can be achieved through a static taint analysis where the taint source is any global memory. In addition, we also consider local variables whose values are indirectly decided by global memory as taint sources (more detail in §6.1). Basically, whenever any source flows into any sink, the sink statement is effectively an UD. The result of this step allows us to quantify how often the dependency challenge impedes the fuzzer to make further progress.

Since accuracy is critical for ensuring that our measurement is not too far off from the ground truth, we have made changes and improvements to a state-of-the-art static analysis engine Dr.Checker [23] to adapt it to our use case and demonstrate decent accuracy. In addition, it is worth noting that the static taint analysis is based on the toolchain of LLVM [21]. Therefore, it is necessary to map an UC address to the LLVM instruction. The details of the above are described in §5.

Step 3: Recognizing Write Statements of Dependencies. Given an UD, next step is to identify WSs throughout a kernel module [1] that can influence the value of the global memory, therefore having a chance to resolve the dependency. In addition, we aim to pinpoint any EWS. To identify WSs, we rely on static alias analysis by searching for WSs that change the same global variable(s) as used in an UD.

[1]The EWSs are possible outside the kernel module. Due the scalability issue, we do not extend the scope of static analysis to the whole kernel, but we will manually search the whole kernel in later manual analysis.

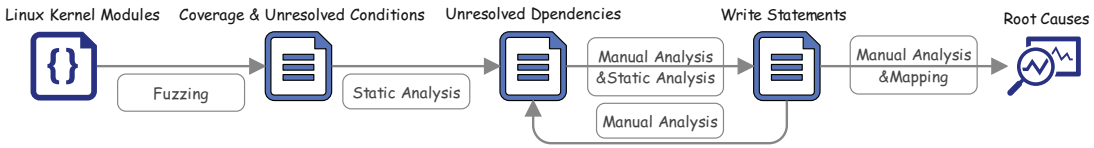

Figure 2: Measurement pipeline including the data collected from dynamic, static, and manual analysis

Once static analysis results provide a set of WSs, we then manually inspect each WS and determine which one(s) are in fact "effective". Unfortunately, this goal is extremely challenging to fulfill automatically because many WSs write values that are dynamically determined, and it can be hard to construct a test case to reach such statements as well. In fact, according to our analysis, the majority of the WSs are simply never even covered in our long fuzzing sessions. Nevertheless, when the write values are statically determined, *i.e.,* constants, our static analysis can determine their effectiveness and prune any WSs that are obviously not effective. For the remaining WSs, we have to resort to a manual analysis to identify EWSs.

Step 4: Distilling Root Causes. Finally, to drill down to the root cause of an UD, we need to prove or disprove whether it can be resolved. Specifically, if we manage to find any EWS from the previous step, we will attempt to construct an actual test case that can exercise the EWS, which will eventually resolve the dependency. We can then analyze why the fuzzer failed to generate such a test case (*i.e.,* root causes) by inspecting its runtime logs recorded previously. However, in practice, we find it extremely challenging to construct such test cases, even for experienced researchers.

First of all, most EWSs were never even exercised during the fuzzing session. It can be challenging to construct a test case that can reach the statement. For example, the EWS may be guarded again by some UCs (and even UDs). Even if we can reach the EWS, it can often be tricky to precisely control the value (which may in turn depend on global memory). This can lead to a recursive analysis starting all the way from the second step (more details in §6.2). Even though our static analysis can help with this process of determining UDs and narrowing down the search space of EWSs, it is still a substantial undertaking.

Second, it turns out that it is challenging to "mechanically" follow the procedure of looking for EWSs and how to reach them. For example, there are often complications such as test cases not being able to reproduce the same consistent behaviors (as some kernel states are modified by the test cases). Therefore, in the end, we end up firstly making sense of the overall architecture of each kernel module, *i.e.,* understanding the semantics of most critical functions and data structures, before jumping into the details of each dependency. This includes manual auditing of the source code as well as dynamic testing, *e.g.,* setting breakpoints and observing behaviors of certain functions. As more details will be discussed in §6.3, on average, each case takes about 0.5 hour to 4 hours to analyze. We believe this is a valuable dataset that will benefit any future research on kernel fuzzing.

5 IMPLEMENTATION

In total, there are 10.9k (C++), 4.7k (Go) and 0.5k (Protobuf) lines of code for the whole pipeline, including the mapping between binary and LLVM bitcode (§5.1), static taint analysis (4k C++ lines of changes to Dr.Checker in §5.2), data sharing (serialization and deserialization) between all components, collecting statistics, organizing information to support manual analysis, and the experimental solution (§7). Below we describe in more detail about the two main components in the measurement pipeline, Mapping (§5.1) and Static Analysis (§5.2). We will open source all of the components and data to facilitate the reproduction of results and future research.

5.1 Mapping Between Binary and LLVM Bitcode

The coverage information collected during fuzzing is about binary instructions, specifically through the instrumentation provided by kcov [18]. In contrast, our static analysis operates on the LLVM bitcode. For example, when determining whether an UC is an UD, it takes a branch instruction in LLVM as input, *i.e.,* `br cond, true_branch, false_branch`. Therefore, we need to map the binary level coverage back to the LLVM bitcode level. This turns out to be a non-trivial process. Specifically, by default, the Linux kernels are compiled with a high optimization level `O2` (this is also necessary for an efficient fuzzing process), the bitcode files generated at an early stage (unoptimized) in the compilation pipeline are very different from the kernel binary in terms of the control flows and boundaries of basic blocks, making it challenging to map between them.

Our solution is to use optimized bitcode files produced at a much *later stage* in Clang (through an undocumented feature), which share the same control flow graph and basic block structure as those in binary. This allows us to obtain an accurate mapping between the binary and IR instructions. Due to the space limit, we do not go into details here. We will however include the solution in the open sourced version of our measurement infrastructure, which may help other projects that need such mappings.

5.2 Static Taint Analysis

Our static analysis engine is built on top of the state-of-the-art Dr. Checker [23], which is designed to be flow-, context-, and field-sensitive to analyze kernel source (translated into LLVM IR). However, since Dr. Checker's built-in alias and taint analyses are not suitable and precise enough for our purposes, we made two categories of changes: (1) adapting it to work in our use cases, and (2) improving its accuracy.

(1) First, since the LLVM bitcode we analyze is compiled with the `O2` optimization level as described above, exotic forms of IR instructions will be generated by Clang, which are not handled

by Dr. Checker (as it compiles the kernel using `O0`). One such instruction is the multi-index GEP that aggregates multiple layers of pointer offset calculations in a single instruction. For example, `getelementptr %struct.foo* @chunky, i64 1, i32 2, i32 3, i32 4` is equivalent to `&chunky[1].f2.f3.f4` (the last three indices 2, 3, 4 represent the field indexes). Second, instead of treating user-controlled syscall arguments as taint sources (as in the case of Dr. Checker), we need to taint any global memory, which, according to its definition in §2.1, can be either (a) explicitly declared global variables or (b) heap memory reachable from global variables. The former is straightforward. The latter can be tricky because sometimes the reachability relationship is established during some setup or initialization phase, which can often be outside of the scope of a kernel module. An example is the `file` or `device` structure that frequently appear as the first argument of a driver's `ioctl()` handler. Such structures may belong to heap memory and are reachable from some global data structures, but such reachability relationship is set up in the generic kernel as opposed to a specific kernel module. Therefore, we simply apply domain knowledge to such function arguments and label them as taint sources accordingly.

(2) One significant improvement we made over Dr. Checker is the pointer arithmetic resolution. Partly due to the `O2` optimization level, we find many more pointer arithmetic operations such as `*(int*)((char*)p + byte_offset)`, which is basically equivalent to `p->f`, where `p` is originally a pointer to a structure type which contains an integer field `f` at the offset `byte_offset`. Therefore, we analyze the definition of structures and map a byte offset to a particular field in the structure. With this improvement, we will not lose track of such points-to relationships. We also handle the widespread container_of() cases in kernel, where the pointer to a container structure is obtained by subtracting an offset from a pointer to an embedded structure within it. These cases can cause difficulties for a static analysis due to the pointer arithmetic and the unawareness of the container structure type. We address this problem by inferring the container type from the context of the container_of() pointer calculation (*e.g.*, the resulting `char*` pointer can be converted to the actual `struct*` type by a later "cast" IR), and then creating the correct container object hosting the original embedded structure. This way, we can precisely maintain the points-to records.

6 EVALUATION

In this section, we first describe the experiment setup and validation of the static analysis results in §6.1. We then present the measurement results about the prevalence of UDs and the complexity involved in analyzing them in §6.2. Finally, we report the results of our root cause analysis to §6.3.

6.1 Experiment Setup and Validation of Methodology

Fuzzing Setup. We pick four Linux kernel driver modules for testing (from kernel version 4.16) as listed in Table 1. They are all available and compiled based on the `defconfig` (default kernel configuration), which represents commonly used drivers. There are three reasons we pick them:

Table 1: Description and fuzzing time for each module

Name	SLoC	Template size (#lines)	Time (hour)
cdrom	4.6k	344	48
snd_seq	14k	278	48
ptmx	39k	324	48
kvm	60k	806	120

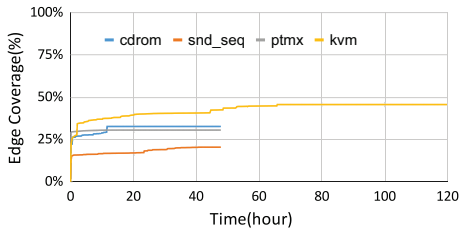

Figure 3: Coverage over time

Table 2: Categories of unresolved dependencies and sample cases for manual analysis

#DomI	#UC	#DomI$_{UC}$	#SampleA	#UD	#DomI$_{UD}$	#SampleR
<11	2182	10,014	74	1415	7,176	35
11 - 100	577	23,553	32	428	17,565	37
101 - 190	124	16,885	10	80	11,073	13
>190	94	62,448	17	65	39,378	30
Sum	2977	112,900	135	1988	75,192	115

- Compared to the core Linux kernel, driver modules account for the majority of the kernel code and therefore the attack surface as well. There are about 22.7M SLoC for the whole Linux kernel but 16.4M SLoC (71.9%) of them are device drivers.
- As mentioned in §3, compared to the less tested and less popular drivers, we choose more popular ones that are relatively well tested and have relatively comprehensive Syzkaller *templates*. At the start of our analysis, there were only 32 drivers with Syzkaller templates, and 15 of which had well-written templates, which limits our choices significantly.
- We aim to cover different types of Linux device drivers — one character device driver: *ptmx*, one block device driver: *cdrom*, and one main sub-category of character device driver (miscellaneous): *kvm*, and one other character device driver that is outside of the /drivers folder: *snd_seq*.

As shown in Table 1, we test each module individually for *at least* 48 hours, repeated three times. The machine we use to conduct fuzzing has an Intel(R) Xeon(R) Gold 6248 CPU and 512GB of RAM, and runs Ubuntu 18.04 LTS. We use 32 CPU cores (in 32 different VMs) when fuzzing a module. We collect the union of coverage achieved in all three fuzzing runs. The fuzzing time is determined experimentally based on how long it took for coverage to converge, as will be shown in Figure 3. For example, the coverage of *ptmx*, *snd_seq*, and *cdrom* still improves after 24 hours and we allocate 48 hours for them. On the other hand, the coverage of *kvm* still improves even after the 72-hour mark, which prompts us to allocate 120 hours for it. This way, we ensure all the remaining UCs at the end of the fuzzing session will likely represent the limitations of Syzkaller and its templates.

Table 3: Accuracy of static analysis

Name	#UC	Manual Analysis			Static Analysis					
		#UD	#WS	#EWS	#UD	#FP	#FN	#WS	#FP	#FN
cdrom	16	11	3.00	1.00	11	9.09%	20.00%	3.00	0.00%	0.00%
snd_seq	16	13	3.00	1.00	8	0.00%	60.00%	3.00	5.56%	5.56%
ptmx	25	17	7.88	1.75	16	0.00%	20.00%	7.50	0.00%	4.76%
kvm	78	45	3.00	1.33	40	0.00%	19.23%	2.94	1.85%	3.70%
Sum/Avg	135/-	86/-	-/4.08	-/1.33	75/-	1.33%	24.39%	-/3.97	1.36%	4.08%

Table 4: Classification of unresolved conditions

Name	#UC	#UD		#unknown	#non-UD
		#Global	#Control		
cdrom	16	7	4	2	3
snd_seq	16	10	3	2	1
ptmx	25	12	5	3	5
kvm	78	29	16	24	9
Sum	135	58	28	31	18

Table 5: Prevalence of unresolved dependencies

Name	#UncoveredE/#E	#UD/#UC	#DomE$_{UD}$/#DomE$_{UC}$
cdrom	615/915 (67%)	37/54 (69%)	80/122 (99%)
snd_seq	7355/9255 (79%)	162/274 (59%)	204/356 (57%)
ptmx	3541/5105 (69%)	254/289 (88%)	763/830 (92%)
kvm	15471/28516(54%)	1554/2050(76%)	4073/5677(72%)
Sum	26982/43791(62%)	2007/2667(75%)	5106/8037(73%)

Sampled Cases for Manual Analysis. We sample two datasets for manual analysis, summarized in Table 2. The first dataset is to validate the accuracy of our static analysis results. The second is for the root cause analysis. For the first dataset, we start from UCs and manually determine which are in fact UDs. Then we also conduct an exhaustive manual search for their corresponding WSs, with the understanding of the semantics of each kernel module (as discussed in §4). Such UDs and their WSs serve as ground truth. To cover a representative set of samples, we categorize the UCs by the number of instructions behind them, *i.e.,* the instructions (denoted as #DomI) that are dominated by the condition in the same function (and that there are no other ways to reach them without satisfying the condition). A smaller #DomI indicates that the condition is likely related to simple error handling, whereas a larger #DomI is related to the functional part of the kernel module. As shown in Table 2, we partition the UCs into four ranges based on #DomI, from which we then randomly sample #SampleA of cases. Note that although the first category (*<11*) happens the most (#UC), but the aggregated #DomI is the lowest, whereas the last category has fewer UCs but more aggregated #DomI. As a result, we slightly favor the last category as they represent more functional code and are generally more worthwhile to test. For the second dataset, we start directly with UDs (as opposed to UCs), so we can investigate their root causes. In total, we randomly sampled 115 UDs over different categories as shown under the #SampleR column in Table 2, favoring the last category even more because resolving such dependencies will unlock more functional code.

Accuracy Validation of Static Analysis. From the first sampled dataset that serves as ground truth, we show the static analysis results in Table 3. As we can see, since we try our best to optimize our analysis to avoid exaggerating the UD problem in kernel fuzzing, the overall accuracy is very good. There are a 1.33% false-positive rate and 24.39% false-negative rate with regards to determining if an UC is an UD. That means that there is likely an underestimate of the dependency issue by our static analysis (as there are more false negatives). There are a 1.36% false-positive rate and at least a 4.08% false-negative rate regarding the WS analysis; note that the false-negative rate is a lower bound because we do not claim to have found the complete set of WSs manually for each UD. In other words, the #WS result overall is also an underestimate.

Unreachable Functions Elimination. Our initial observation of the fuzzing results indicate that most kernel modules have much more code than what we can cover with Syzkaller because of obviously unreachable functions. As a result, we prune unreachable functions as described later in §8 because otherwise the percentage of code coverage in the end will look unrealistically small, which may leave us with the wrong impression about the kernel fuzzing performance.

Classification of Unresolved Conditions. We now give an overview of the 135 UCs from the SampleA dataset. Based on our manual characterization of these cases, as shown in Table 4, we can divide them into 86 (58+28) UDs, 18 non-UDs, and 31 unknown cases. We can further break down the UD cases into 58 cases where the UD is directly affected by a global memory and 28 cases where the UD is indirectly affected by global memory. The latter cases represent UDs where their conditions check the value of a (local) variable whose value is indirectly decided by global memory through control dependence (as opposed to data dependence). For example, "`if(UD) var=true; if(var)//Uncovered code`". `if(var)` is classified as UD because the local variable `var` is indirectly affected by `if(UD)`. For the non-UD cases, they always correspond to conditions whose values are influenced purely by syscall arguments. Finally, for the unknown cases, they represent cases that are difficult to analyze because they require analysis of either the assembly code or code external to the driver module itself. We exclude these unknown cases from our subsequent analysis. Overall, the UD cases represent 86/104 = 83% of the UC cases, indicating that the dependency challenge is indeed a major hurdle for kernel fuzzing.

6.2 Measurement Results

In this section, we report (1) static analysis results at scale, and (2) quantitative metrics that show the difficulty of analyzing these UDs.

Prevalence of Unresolved Dependencies The results are shown in Table 5. The number of total edges (#E) represents the total possible coverage (from one basic block to another) for each module. After taking the union of coverage obtained from three fuzzing runs, we report the remaining uncovered edges (#UncoveredE). As we can see, even with all the pruning of unreachable code, there are still many uncovered edges even after 3×48 hours or 3×120 hours of fuzzing for each module, ranging from 54% to 79% percentage-wise. Out of edges that are never covered, we further look at those with corresponding conditions that are exercised but never evaluated to the desired value (*i.e.,* either true or false). These correspond

Table 6: Write statements stats from static analysis

Name	#UD	Static Analysis			
		#WS	#WSC	#WSE	#WSE/#WS
cdrom	37	1.90	1.45	0.46	24.21%
snd_seq	162	3.56	1.03	2.53	71.07%
ptmx	254	18.05	5.50	12.54	69.47%
kvm	1554	13.51	4.53	8.99	66.54%
Sum/Avg	2007/-	-/13.25	-/4.36	-/8.90	67.15%

Table 7: Root causes of unresolved dependencies

Name	#UD	#DeadCode §6.3.1	#Environment §6.3.2	#Unobserved §6.3.3	#Template §6.3.4	#Search §6.3.5	unknown
cdrom	7	0	6	1	0	0	0
snd_seq	32	16	2	9	2	3	0
ptmx	33	12	5	1	5	5	5
kvm	43	7	14	0	15	1	6
Sum	115	35	27	11	22	9	11

to UCs (#UC), which are much smaller than the total number of uncovered edges, but each of them acts as a guard that prevents many more subsequent edges from being covered. Out of these UCs, our static analysis reports on average 75% as UDs (#UD). Even though dependency is a known problem, this prevalence of it is still surprising (keep in mind that the number is likely an underestimate). According to the breakdown by modules, *ptmx* and *kvm* have the higher #UD. This is expected because the more complex a module is, the more global states are likely introduced and hence more dependencies. #DomE$_{UC}$ describes the sum of the number of uncovered edges that are intra-procedurally dominated by the guard conditions (reachable only from the UCs), and #DomE$_{UD}$ is the sum over all the UDs. We choose intra-procedural domination because we do not want to overestimated it. It shows that UDs roughly lock away three quarters of the code.

Write Statements. From our static analysis results, we find that the average number of WSs for each UD is about 13, as shown in the #WS column in Table 6. If we look at the number for each module, *kvm* and *ptmx* exhibit many more WSs. In addition, we find that sometimes an UD may involve more than one piece of global memory (*e.g.*, influenced by multiple global variables), and each of them may have multiple WSs.

The next step in the measurement pipeline is to determine which WSs are effective. As discussed in §4, this step is very challenging even with manual analysis. One of the reasons is that these WSs are often writing values determined at runtime (in the form of expressions). As Table 6 shows, 67.15% of the written values are expressions (#WSE) instead of constants (WSC). In the end, as shown in our later manual analysis results in Table 3, we find that the number of EWSs (#EWS) is far smaller than #WS (often only a single one).

The results show that it would be likely an expensive search to exhaustively test every single WS automatically, given that we may need to determine the possible values that can be written (which can be challenging by itself). This problem is exacerbated when we consider the recursive nature of dependencies, as will be shown next.

Recursive Dependencies. Whenever a WS cannot be reached by any of the existing test cases attempted by Syzkaller, it is likely that it may be blocked off due to additional UDs, leading us to recursively analyze more WSs and dependencies. When this happens, the search space can blow up quickly depending on (1) the number of WSs we have to attempt at each depth level which we know is 13 on average and (2) the depth of recursion (difficult to measure). We measure two approximate metrics to quantify the recursive dependency challenge. First, we find that on average 35.91% of #WS are uncovered after the fuzzing sessions are done across all four drivers. This indicates that it requires work to construct a test case before we can even verify whether these WSs are feasible. Second, when the WSs are not covered, we find surprisingly 93.72% of them are due to recursive dependencies based on our static analysis of the dominating conditions. *In other words, if a fuzzer has not managed to reach the EWSs after a long fuzzing session, it is almost always never due to barely missing the opportunity, indicating some more fundamental roadblocks as we will investigate later in §6.3.*

Non-self-contained (Unstable) Test Cases. During the process of our manual investigation of root causes, we find another common hurdle. That is, the test cases generated by Syzkaller are not self-contained. Specifically, Syzkaller (or fuzzers based on it) generates a stream of test cases and executes them, accumulating significant kernel state changes as it progresses (until a reboot occurs). This means that the "success" of a test case can be dependent on the previous ones that may have accidentally set up the kernel state, effectively making these test cases non-self-contained or "unstable". This is important for our manual investigation because we rely on the test cases generated by Syzkaller to reproduce the results, *e.g.*, if a test case is reported to reach some UD, we will re-execute the test case and hope that it will still be able to, otherwise we have to put more effort in the steps mentioned in §4. Unfortunately, a large fraction of test cases that were previously able to reach UDs and WSs become unstable when tested later in isolation (average 97% and 46% respectively). This is another reason why manual analysis can be expensive, as we need to reconstruct the state which can help us understand the root cause of the unresolved dependency.

6.3 Analysis of Root Causes

Overall, it took about 300 person-hours, with the help of all the results produced by automated analysis in the measurement pipeline, in order for us to be confident about the correctness of our results. Even then, there are a few cases where we cannot determine the root causes even after hours of investigation. This means that we cannot either construct a test case that can resolve the dependency or prove that it is impossible. Given the level of difficulty in conducting such an analysis, we will publicly share the datasets, which we believe is valuable to researchers who aim to improve kernel fuzzing.

Overall, we distill and summarize the root causes into six categories (and an unknown category). The results are presented in Table 7. We next describe them one by one.

6.3.1 Dead Code. The amount of dead code in a large complex piece of software such as Linux is a mystery. From our results, we are surprised to see a substantial portion of the UDs (35/115) turn out to lead to dead code, representing almost 30% of UDs. In other

```
1  static struct snd_seq_queue *queue_list[SNDRV_SEQ_MAX_QUEUES];
2  static int seq_free_client1(...) {
3    snd_seq_queue_client_leave(client->number);
4    snd_seq_queue_client_termination(client->number);
5  }
6  void snd_seq_queue_client_leave(int client) {
7    for (i = 0; i < SNDRV_SEQ_MAX_QUEUES; i++) {
8      if ((q = queue_list_remove(i, client)) != NULL)
9        queue_delete(q);
10   }
11 }
12 void snd_seq_queue_client_termination(int client) {
13   for (i = 0; i < SNDRV_SEQ_MAX_QUEUES; i++) {
14     if ((q = queueptr(i)) == NULL)
15       continue;
16     spin_lock_irqsave(&q->owner_lock, flags);
17     if (q->owner == client)
18       // Uncovered branch
19       q->klocked = 1;
20     spin_unlock_irqrestore(&q->owner_lock, flags);
21     if (q->owner == client) {
22       // Uncovered branch
23       if (q->timer->running)
24         snd_seq_timer_stop(q->timer);
25       snd_seq_timer_reset(q->timer);
26     }
27     queuefree(q);
28   }
29 }
```

Figure 4: Case of dead code

words, it is simply impossible to reach these uncovered branches behind UDs no matter how hard we try. The best thing fuzzers can do it to recognize them and avoid wasting time trying to cover them.

Dead code is a well known issue and compilers routinely perform dead code elimination [5, 9]. We suspect that the reason why compilers fail to identify them is due to the nature of UDs requiring the analysis of global memory (beyond a local context). For example, we find a function may double check the existence of certain elements in a global queue right after its caller removes them (the example is available in the uploaded dataset, among other examples). To get further confirmation, we reported the case to Linux kernel developers and they have agreed with our assessment and eliminated the dead code subsequently. Even though we have not reported all the cases to Linux, we exercise the same rigor across all the dead code cases.

Interestingly, except *cdrom*, all other kernel modules have a substantial fraction of dead code cases out of their UDs. For *snd_seq*, half of its UDs are surprisingly dead code (16 cases out of 32). In addition, we are curious to see whether the dead code cases only represent small pieces of code (*e.g.,* redundant error checks). In general, the data are indeed in line with the hypothesis, especially given the way these cases are sampled (shown in Table 2). However, there is still a substantial fraction of counterexamples. Following the categorization in Table 2, we find 18 out of the 35 dead code cases have only a small number of dominated instructions in the same function (*<11*). However, the remaining 17 cases are spread out: 10 cases with *11-100* dominated instructions, 4 cases with *101-190* dominated instructions, and 3 cases with *>190* dominated instructions.

We give a fairly complex dead code example that we manually determined in Figure 4. There are two unresolved dependencies (line 17 with 23 uncovered instructions and line 21 with 61 uncovered

```
1  static int cdrom_read_cdda(...) {
2    if (cdi->cdda_method == CDDA_OLD)
3      // Uncovered branch
4    ... }
5  int register_cdrom(struct cdrom_device_info *cdi) {
6    if (cdi->disk) cdi->cdda_method = CDDA_BPC_FULL;
7    else cdi->cdda_method = CDDA_OLD; // Uncovered branch
8    ... }
9  static int ide_cd_probe(...) {
10   if (drive->media != ide_cdrom) goto failed;
11   devinfo->disk = info->disk;
12   register_cdrom(devinfo); ... }
13 static int probe_gdrom(...) {
14   if (gdrom_execute_diagnostic()!=1) return -ENODEV;
15   register_cdrom(gd.cd_info); ... }
```

Figure 5: Case of environment dependency

instructions). Based on the control flow graph and data flow graph, it does not seem like the uncovered branches of line 18-19 and line 22-25 are dead code. However, if we look at the call graph, we can find that the only caller of the function `snd_seq_queue_client_termination()` is the function `seq_free_client1()` that calls the function `snd_seq_queue_client_leave()` before the function `snd_seq_queue_client_termination()`. What the function `snd_seq_queue_client_leave()` does is to remove all the elements in global queues if their client IDs are equal to `client->number` (see line 3). Interestingly, function `snd_seq_queue_client_termination()` goes through the same global queues to look for elements that have a specific client ID (again `client->number` as shown in line 4). This will obviously lead both conditions at 17 and 21 to always evaluating to false. To get further confirmation, we reported this case to Linux kernel developers and they have agreed with our assessment and eliminated the dead code subsequently. Even though we have not reported all the cases to Linux, we exercise the same rigor across all the dead code cases.

6.3.2 Environment Dependency. We find that oftentimes an UD may depend on the configuration of and values from the execution environment (*e.g.,* hardware). If the underlying execution environment is not the expected type or returns the expected result, a dependency will not be resolved. Note that we do attempt to prune code that obvious cannot be reached because of environment dependency (see §8). Nevertheless, our heuristics only focused on function pointers which point to different targets depending on the underlying execution environment. Other than those, we find there are still many other cases (most are much smaller) that are not excluded earlier. An analysis that could automatically recognize them would be helpful so users of fuzzer can tune the configuration.

We show an example in Figure 5, where the UD is `cdi -> cdda_method == CDDA_OLD` (line 2). We find the EWS `cdi -> cdda_method = CDDA_OLD` (line 7), which is not covered because of yet another UD `if (cdi->disk)` (line 6). The value of `cdi->disk` is decided by the caller of `register_cdrom()`, which can be either `ide_cd_probe()` (line 9) or `probe_gdrom()` (line 13). These two functions check the type of the hardware device, and either writes to `devinfo->disk` (line 11) or does not. We omit a few other device types and the exact hardware read functions for brevity. Only if we have the correct hardware device present (in this case a GDROM, not CDROM), will we reach line 7 and in turn line 3.

```
1  static struct snd_seq_client *clienttab[SNDRV_SEQ_MAX_CLIENTS];
2  static int open(*file) {
3    clienttab[client->number] = client;
4    client->type = USER_CLIENT;
5    file->private_data = client; }
6  static long ioctl(*file, cmd, arg) {
7    *client = file->private_data;
8    snd_seq_ioctl_create_port(client); ... }
9  static int snd_seq_ioctl_create_port(...) {
10   if (client->type == KERNEL_CLIENT)
11     // Uncovered branch
12   ... }
13 int snd_seq_create_kernel_client(...) {
14   client->type = KERNEL_CLIENT;
15   return client->number; }
16 int __init snd_seq_system_client_init()) {
17   sysclient = snd_seq_create_kernel_client(NULL,0,"System");
18   snd_seq_ioctl_create_port(sysclient); ... }
19 module_init(alsa_seq_init)
```

Figure 6: Case of unobserved dependency (module *init*)

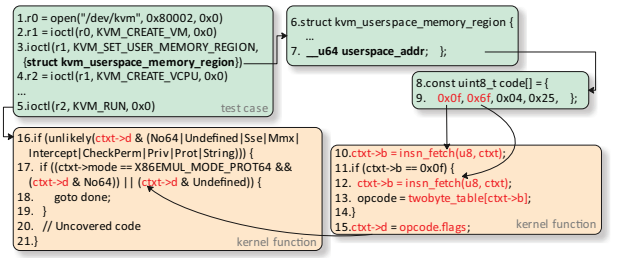

Figure 7: Case of unresolved dependency because of incomplete templates

In total, we identify 27 such cases as shown in Table 7, which represent almost a quarter of all UDs. Among them, we find that 18 are about reading the hardware type or property (nine in module `init` functions, and nine in syscalls, a special situation here is nested guests in *kvm*, which counts six cases). Six of them attempt to read something that can change at runtime. The remaining three are undetermined.

6.3.3 (Partially) Unobserved Dependency. This is an interesting category (although rare in our investigation) related to the design of Syzkaller where certain functions are simply not in the scope of the fuzzer. For example, any code that occurs in the module *init* or *exit* functions, or the bottom-half processing [27] are not tracked by Syzkaller. This means that any code that is reachable from either category of functions will never be covered (even if they do get executed). This is because such coverage cannot be attributed to a specific test case. This leads to a variety of problems.

First, this can lead to incorrect accounting of UDs. Specifically, some functions can be reachable from both syscall entry points and the above categories. We find that it is possible that such functions have UDs during syscall fuzzing, but in reality the dependency is in fact resolved during these other unobserved execution paths (from *init*/*exit* or bottom-half). For example, `snd_seq_create_kernel_client()` in Figure 6 contains the only EWS (line 14) for the UD (line 10), and is only called by the *module init* function, (*i.e.,* `__init alsa_seq_init` at line 19). When the UD is reached from syscalls, *i.e.,* `ioctl()`, the kernel state `client->type` always takes the value of `USER_CLIENT`, which is set by the WS at line 4 in `open()`. Obviously, this will lead the Syzkaller to think that it is unresolved. There are two cases in total that can be essentially considered resolved dependencies.

Second, because Syzkaller was not aware of functions related to the *init*/*exit* and bottom half, it also does not have the ability to schedule them during fuzzing. This leads to EWSs that are invoked only sporadically (in the case of the bottom half) or only at the module load time or unload time. For example, the module init may initialize some kernel state to an expected value (by an EWS). However, Syzkaller may first schedule test cases that accidentally overwrite the expected value before the UD is even seen. We find nine such cases where the only EWSs are located in `init/exit` functions or the bottom half. There are two cases under this category where the dependencies are not triggerable.

As we can see, this is highly dependent on the kernel modules. Even though rare overall, they do constitute a non-negligible portion of the cases in *cdrom* (14.29%) and *snd_seq* (28.13%). We anticipate seeing many more cases of bottom-half processing in modules with frequent external interactions, *e.g.,* audio, camera, and network drivers.

6.3.4 Incomplete Templates. The root causes introduced thus far are outside of the control of a fuzzer (*e.g.,* dead code and environment dependencies). Now we move on to the root causes that technically fall under the scope of Syzkaller. Specifically, we mentioned that Syzkaller relies on templates that encode the knowledge about the syscall interface for each kernel module. The quality of templates has a direct impact on the fuzzing performance, as the test cases are generated based on the templates. Even though the templates are already comprehensive, as suggested by their sizes shown in Table 1, from our analysis, we discover a variety of deficiencies. This can include missing knowledge about syscall relationships, argument types and ranges, etc., which result in failures to reach otherwise reachable EWSs. Basically, if an UD can be resolved clearly with an amended template, it belongs to this category. Overall, our manual investigation reports 22 cases under this category (#Template) as shown in Table 7. This represents the largest category of root causes that will allow coverage improvement by resolving more dependencies.

Most cases belong to *kvm*, which is the most complex piece of kernel module out of the four we analyze. An example is shown in Figure 7. The UD is at line 17, and we locate the EWSs at line 15. As we can see, the value of the `ctxt->d` is determined by another variable `opcode.flags`, which is in turn determined by the WS `opcode = twobyte_table[ctxt->b]` at line 13. Finally, we know that the data (an instruction buffer) referenced by `userspace_addr` (which is a pointer field in an argument of the syscall at line 3) can decide the value of `ctxt->b`. However, the template in Syzkaller does not correctly describe what the buffer should look like. In order to resolve the dependency, a test case would need a specific sequence of instructions in the buffer as described in lines 8 and 9. Unfortunately, Syzkaller is simply not aware of the desirable byte sequences. In total, we find about 11 cases that are similar to this example in *kvm* modules.

There are also similar but simpler examples where a single argument needs to take a magic number, much like what commonly occurs in userspace programs. In the kernel modules we analyzed, we observe eight such cases.

In addition, we observe that the templates can also miss explicit dependencies. Across the four drivers though, there is only a single case we find in *kvm*. This confirms our assertion that the templates written for these four drivers are already relatively comprehensive.

6.3.5 Specialized Search Requirement. Finally, there are a few cases that need a more specialized search algorithm that is beyond the capability of a general-purpose fuzzer. Our manual analysis finds 9 cases related to the search algorithm (#Search) out of 115 in Table 7.

- **Interleaving of Multiple Threads:** In order for the UD to read the expected value, sometimes a race condition is required. In other words, a specific thread interleaving has to occur in order for the value written by the EWS to be exposed. If the window of opportunity is small, it is highly unlikely that Syzkaller will be able to resolve the dependency. In total, we find six cases.
- **Specific Sequence of syscalls:** Some EWSs need multiple iterations to successfully change the kernel state to the desired value (*e.g.,* `++i`). This means that one needs to repeatedly invoke one or more syscalls that contain the EWSs to resolve the dependency. In total, we find three cases related to the sequence of syscalls.

7 FUTURE RESEARCH DIRECTIONS

From the previous results, we have shown that UDs are a prominent reason for uncovered code in kernel fuzzing. We also show they are challenging to resolve due to a diverse set of root causes. In this section, based on the insights we gained from our measurement, we summarize two main future research directions that have to promise to overcome these challenges.

Open Problem: Cross-Syscall Input Propagation Analysis. As reported in §6.3.4, 19% of the UDs are actually the result of imperfect templates (missing certain syscall argument knowledge). This means that once the templates are amended, the dependencies can be potentially resolved (and we verify that it is indeed the case with experiments). In this section, we propose a new analysis that tracks the propagation of a syscall input argument across syscall invocations.

We wish to point out this is a unique type of dependency because the WS always writes a value specified in a syscall argument. If we blindly target any dependency without understanding its unique pattern, it is unclear what we should do to resolve them. For example, one can attempt a whitebox approach similar to HFL [19] to locate the write and read pairs regarding the same global memory. However, as we showed in our measurement (see §6.2), there are typically several possible WSs and it is unclear which one is the EWS. This is likely why HFL has focused on only explicit dependencies where typically there is only one WS. To understand how this general direction will work in practice, we actually developed a failed solution (prior to having the full insight from our measurement study) that attempts to collect all the write syscalls during fuzzing, and then pair them up with the read syscalls. The results show that whatever implicit dependencies that we are able to resolve in this solution, Syzkaller can also resolve them, in most cases even quicker. This is a good indication that Syzkaller is actually already sophisticated enough in terms of its algorithmics. As long as the templates have encoded sufficient knowledge regarding syscalls and argument types and possible values, given sufficient fuzzing time, Syzkaller will be able to resolve the implicit dependencies that are "covered" by the templates. Unfortunately, it does not resolve the hard dependencies such as those that are reported in our results. This is because we simply do not have the knowledge of what correct syscall arguments to supply (not given in syscall templates) in order for the WS to write the expected value. Thanks to our measurement insight as shown in §6.3.4, we know that 19 cases out of 22 are due to incomplete descriptions of syscall arguments.

Secondly, we observe that more than half of such inputs (11 cases) are not processed immediately in a single syscall invocation. Instead, they are stored in some global memory in one syscall, and then used in another syscall invocation. If we analyze the syscall where the use happens, oftentimes they look just like a magic number check (`if(g_array[0] == MAGIC_NUM)`, which should be theoretically easy to resolve (*e.g.,* through symbolic execution). Nevertheless, these magic number checks are performed against the global memory (to make which symbolic directly is meaningless) rather than syscall inputs. Those cases would become solvable if there is a cross-syscall input propagation analysis (in the form of either static or symbolic) which could find the related inputs. We have leveraged our static taint analysis developed for the measurement and found that indeed such cross-syscall propagations can be identified. They have resulted in us identifying four missing descriptions in the templates (two of which have been fixed in a later version of Syzkaller). Therefore, we believe a cross-syscall input propagation analysis is an effective method to resolve such complex dependencies.

Open Problem: Multi-Interface Fuzzing. As we discussed in §6.3.3, we show that many dependencies can be resolved only when we consider the interfaces that are not out of the scope of Syzkaller (*i.e.,* unobserved). We have given two specific categories of interfaces including hardware-side input and the module `init/exit` functions. We have mentioned that the latter should be included as part of the syscall fuzzing interface. On the other hand, hardware-side input has been considered in recent works [26, 28, 33].

We wish to point out that this is a very different problem from the current hardware-side fuzzers such as USBFuzz [26] and Frankenstein (Bluetooth fuzzing) [28], which focus primarily on the attack surface from the hardware side only. This is understandable because these drivers do process complex inputs from the hardware (in addition to those from syscalls). Nevertheless, from the insight gained from our measurement, we show syscall fuzzing and hardware-side fuzzing can be intertwined. That is, in order to make further progress in syscall fuzzing (*i.e.,* covering certain branches), we actually need proper hardware-side input to arrive at the right time to write to the global memory with the expected value. To generalize this observation, conversely, when performing hardware-side fuzzing, we might encounter dependencies that can be resolved by syscall inputs only (*e.g.,* putting the device into certain states by syscalls).

We frame the goal as multi-interface fuzzing as it needs to coordinate the inputs from multiple different types of interfaces (*e.g.,* inputs from syscall and hardware side can be interleaved). We believe this is a worthwhile research direction as the bugs that are

uncovered this way are likely hidden deeply in some difficult-to-reach driver states.

8 DISCUSSION AND LIMITATION

Limited Data Set Scale. In this paper, we delve into four specific Linux kernel modules to investigate the dependency challenge. Even though we believe they are representative of kernel modules of varying type and size, their functionalities may not be diverse enough to cover other modules such as GPU drivers, network modules, file systems, etc.. Nevertheless, based on our observation, the distilled dependency challenges do appear general enough for other drivers. In the future, we plan to pick more modules and characterize their differences in terms of their respective dependency challenges. In addition, much of the analysis is conducted manually which is hard to scale. Nevertheless, we plan to expand the analysis effort as future work to improve kernel fuzzing.

Unreachable Functions Elimination. As we mentioned in §6.1, we prune as much unreachable functions as we can. Specifically, we prune two types of unreachable functions: (1) interrupt handling related functions that can not be reached from userspace, and (2) functions that are reachable only when certain hardware environment is present. For the first case, based on the call graph generated during static analysis, we prune all the functions that cannot be reached from syscall entry points, *e.g.,* `ioctl()`. To account for potential inaccuracies in call graphs (due to indirect calls), we apply a known type-based method [6] to conservatively generate call graphs, *i.e.,* as long as the signature of the function pointer is compatible with that of a target function, we will consider it a valid target. This method guarantees that all potential targets will be discovered and no functions will be incorrectly pruned (this method is commonly used to enforce control-flow integrity [6]). The second case turns out to be much more challenging as hardware-dependent code can be scattered throughout a module. As a first order approximation, we inspect all function pointers and their potential targets and look for cases where the target is dependent on the underlying hardware environment. A major example is the *svm* part of the `kvm` module, which is dedicated to AMD CPUs, containing 2,889 basic blocks. In total, the two methods can prune about 33% of edges across modules on average.

Accuracy and Benefit of Manual Analysis. Our measurement studies heavily relied on the manual analysis to identify the root causes of dependencies. This is a non-trivial task because every dependency may look different and unique in certain aspects. To ensure that we do a good job, we not only look at the UDs themselves. Instead, we read the overall structure of the whole kernel module to understand its design in a big picture, which allows us to make an accurate assessment of the root causes. Unfortunately, it is very much difficult to replace our manual analysis with automated program analysis, which means fuzzer can not resolve those hard problems automatically like humans. However, fuzzers have their own ways that conduct a "random" search for test cases in a given scope. And the root causes from manual analysis can help either set up the fuzzing scope or offer better search.

Generalization to general software fuzzing We aim not to overclaim our findings beyond kernel fuzzing, as the OS kernel is an important class of software that deserves attention by itself. Nevertheless, we believe that fuzzing stateful and multi-entry programs, especially device drivers, will likely encounter similar challenges, although the distribution of the root causes might differ.

9 RELATED WORK

Linux Kernel Fuzzing. In recent years, there is a plethora of research on improving kernel fuzzing with the goal of achieving more coverage and finding more bugs. kAFL [30] supports x86 based kernels and speeds up fuzzing by hardware-assisted feedback. Razzer [17] leverages static and dynamic analysis techniques to find race bugs through fuzzing. There are also tools built on fuzzing specific Linux kernel subsystems, including file systems [20], device drivers (by mutating hardware inputs) [33], and hypervisors [29]. The work [31] deploys fuzzing to the enterprise-level Linux kernel. Nevertheless, few address the dependency challenge directly.

Stateful Fuzzers. There are some stateful fuzzers which try to consider states during fuzzing, for user space programs, particularly network protocols. RESTler [1] uses a lightweight static analysis to explore service states of REST API. And the work [12] tries to investigate how to extend stateful REST API fuzzing (*e.g.,* RESTler) in general. SPFuzz [32] uses the knowledge from the RFC to help the fuzzing. SNOOZE [2] needs humans to define the states of protocols to assist fuzzing. Steelix [22] uses program-state, which includes coverage information and comparison progress information, to guide fuzzing. The work [8] improves fuzzing of TLS by analyzing its state machine, which again requires significant manual effort. The work [34] generates test cases for a compiler when there are well-defined syntax and non-trivial semantics.

Static Taint Analysis on Linux kernels. There are several static taint (or data flow) analysis tools against Linux kernel source. Dr. Checker [23], UBITect [36] and K-miner [11] are based on the LLVM framework [21] and are all open source. PacketGuardian [3] is another one based on CIL [24]. Dr. Checker best suits our need and our static analyzer is built on top of it.

10 CONCLUSION

In conclusion, we have conducted an in-depth investigation of the dependency challenge in kernel fuzzing, using a combination of static and manual analysis. With a comprehensive set of data, we demonstrate the challenges when analyzing the unresolved dependencies in Linux kernel. In addition, we distill the root causes of unresolved dependencies. Finally, we reveal many unexpected observations and point to new research directions that will improve kernel fuzzing.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grants CNS-1953933 and CNS-1953932.

DATA AVAILABILITY

Source code and datasets related to this article can be found at https://www.doi.org/10.5281/zenodo.5348989 and https://www.doi.org/10.5281/zenodo.5441138.

REFERENCES

[1] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: stateful REST API fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 748–758. https://doi.org/10.1109/ICSE.2019.00083

[2] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin C. Almeroth, Richard A. Kemmerer, and Giovanni Vigna. 2006. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4176)*, Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel (Eds.). Springer, 343–358. https://doi.org/10.1007/11836810_25

[3] Qi Alfred Chen, Zhiyun Qian, Yunhan Jack Jia, Yuru Shao, and Zhuoqing Morley Mao. 2015. Static Detection of Packet Injection Vulnerabilities: A Case for Identifying Attacker-controlled Implicit Information Leaks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 388–400. https://doi.org/10.1145/2810103.2813643

[4] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. 2021. SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security CCS*.

[5] Yih-Farn Chen, Emden R. Gansner, and Eleftherios Koutsofios. 1997. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *Software Engineering - ESEC/FSE '97, 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering, Zurich, Switzerland, September 22-25, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1301)*, Mehdi Jazayeri and Helmut Schauer (Eds.). Springer, 414–431. https://doi.org/10.1007/3-540-63531-9_28

[6] Clang. 2021. *Control Flow Integrity Design Documentation.* https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html

[7] 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, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2123–2138. https://doi.org/10.1145/3133956.3134069

[8] Joeri de Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 193–206. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

[9] Saumya K. Debray, William S. Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.* 22, 2 (2000), 378–415. https://doi.org/10.1145/349214.349233

[10] LTP developers. [n.d.]. *Linux Testing Project.* https://linux-test-project.github.io

[11] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering Memory Corruption in Linux. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_05A-1_Gens_paper.pdf

[12] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 725–736. https://doi.org/10.1145/3368089.3409719

[13] google. 2021. *ClusterFuzz.* https://github.com/google/clusterfuzz

[14] google. 2021. *syzbot.* https://syzkaller.appspot.com/upstream

[15] google. 2021. *syzkaller.* https://github.com/google/syzkaller

[16] 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, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2345–2358. https://doi.org/10.1145/3133956.3134103

[17] 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 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 754–768. https://doi.org/10.1109/SP.2019.00017

[18] kernel. 2020. *kcov.*

[19] 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 2020 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2020.24018

[20] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 147–161. https://doi.org/10.1145/3341301.3359662

[21] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[22] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 627–637. https://doi.org/10.1145/3106237.3106295

[23] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 1007–1024. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry

[24] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2304)*, R. Nigel Horspool (Ed.). Springer, 213–228. https://doi.org/10.1007/3-540-45937-5_16

[25] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 729–743. https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[26] Hui Peng and Mathias Payer. 2020. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2559–2575. https://www.usenix.org/conference/usenixsecurity20/presentation/peng

[27] Alessandro Rubini and Jonathan Corbet. 2001. *Linux device drivers.* " O'Reilly Media, Inc.".

[28] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. 2020. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 19–36. https://www.usenix.org/conference/usenixsecurity20/presentation/ruge

[29] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2020. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/hyper-cube-high-dimensional-hypervisor-fuzzing/

[30] 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 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[31] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. 2019. Industry practice of coverage-guided enterprise Linux kernel fuzzing. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 986–995. https://doi.org/10.1145/3338906.3340460

[32] Congxi Song, Bo Yu, Xu Zhou, and Qiang Yang. 2019. SPFuzz: A Hierarchical Scheduling Framework for Stateful Network Protocol Fuzzing. *IEEE Access* 7 (2019), 18490–18499. https://doi.org/10.1109/ACCESS.2019.2895025

[33] 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 *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.

[34] Vasudev Vikram, Rohan Padhye, and Koushik Sen. 2021. Growing A Test Corpus with Bonsai Fuzzing. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 723–735. https://doi.org/10.1109/ICSE43902.2021.00072

[35] Guangliang Yang, Jeff Huang, and Guofei Gu. 2018. Automated Generation of Event-Oriented Exploits in Android Hybrid Apps. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA,*

February 18-21, 2018. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_04B-3_Yang_paper.pdf

[36] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. 2020. UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 221–232. https://doi.org/10.1145/3368089.3409686