StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing

Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, and Ming Yuan, *Institute for Network Science and Cyberspace / BNRist, Tsinghua University;* Wenyu Zhu, *Department of Electronic Engineering, Tsinghua University;* Zhihong Tian, *Guangzhou University;* Chao Zhang, *Institute for Network Science and Cyberspace / BNRist, Tsinghua University and Zhongguancun Lab*

https://www.usenix.org/conference/usenixsecurity22/presentation/zhao-bodong

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing

Bodong Zhao[1], *Zheming Li*[1], *Shisong Qin*[1], *Zheyu Ma*[1], *Ming Yuan*[1], *Wenyu Zhu*[2], *Zhihong Tian*[3], *Chao Zhang*[1,4*]

[1]*Institute for Network Science and Cyberspace / BNRist, Tsinghua University.*
[2]*Department of Electronic Engineering, Tsinghua University.*
[3]*Guangzhou University.* [4]*Zhongguancun Lab.*

Abstract

Coverage-guided fuzzing has achieved great success in finding software vulnerabilities. Existing coverage-guided fuzzers generally favor test cases that hit new code, and discard ones that exercise the same code. However, such a strategy is not optimum. A new test case exercising the same code could be better than a previous test case, as it may trigger new program *states* useful for code exploration and bug discovery.

In this paper, we assessed the limitation of coverage-guided fuzzing solutions and proposed a state-aware fuzzing solution `StateFuzz` to address this issue. First, we model program states with values of *state-variables* and utilize static analysis to recognize such variables. Then, we instrument target programs to track such variables' values and infer program state transition at runtime. Lastly, we utilize state information to prioritize test cases that can trigger new states, and apply a *three-dimension feedback* mechanism to fine-tune the evolutionary direction of coverage-guided fuzzers. We have implemented a prototype of `StateFuzz`, and evaluated it on Linux upstream drivers and Android drivers. Evaluation results show that `StateFuzz` is effective at discovering both new code and vulnerabilities. It finds 18 unknown vulnerabilities and 2 known but unpatched vulnerabilities, and reaches 19% higher code coverage and 32% higher state coverage than the state-of-the-art fuzzer Syzkaller.

1 Introduction

Fuzzing has become the most popular and effective solution for discovering vulnerabilities, and is widely studied by the industry [20, 27, 40, 53] and the academic community [22, 30, 31, 51]. For example, the OSS-Fuzz project [41] by Google continuously tests 35 open-source projects and has found over 25,000 bugs[1] as of January 2021. In general, fuzzers randomly generate test cases and execute target programs with these test cases. To deal with intrinsic randomness, a large number of fuzzers follow the steps of AFL [20, 53] and utilize code coverage to guide the exploration process of fuzzing. In general, they prioritize test cases that hit new code (i.e., contribute to code coverage) and use them as starting points for further exploration.

Despite the great success, coverage-guided fuzzing solutions also have many limitations. The most critical limitation is that such solutions are code-coverage-centric and are insensitive to other feedback when exploring the test case space. In practice, a large number of programs (including device drivers and network services, etc.) have complicated internal program states and will not continue execution or crash if a specific state is not reached. For instance, a device will not work if a specific status register is not set to an expected value. To test such programs efficiently, a fuzzer should be aware of program states and explore the state space smartly.

Recent works have shed light on exploring program states. For instance, IJON [4] utilizes different forms of manually-provided state representations (e.g., positions in a maze game) to perform not only fuzzing but also gaming like Super Mario. InsvCov [19] uses the likely invariants of programs as boundaries to partition the program state space. AFLNet [38] uses servers' response code as program states to drive network protocol fuzzing. In addition, StateAFL [32] identifies program states by performing locality-sensitive hashing on specific process memory. More research efforts are needed in this direction. In general, there are three questions to answer when developing a state-aware fuzzing solution.

First, what are program states? Essentially, a program state is the execution context of a program, including values of all program variables (from the perspective of software) and values of all memory and registers (from the perspective of hardware). However, the number of such states is overwhelmingly large, and it is hard to track all of them in practice. Thus, a practical fuzzer has to focus on a subset of program states, as IJON and AFLNet did. Moreover, which states are crucial for fuzzing and how to reduce the state space are still open questions.

Second, how to recognize program states and track them

*Corresponding author: chaoz@tsinghua.edu.cn
[1] https://google.github.io/oss-fuzz/#trophies

during fuzzing? IJON relies on manual annotations to mark states and manual program instrumentation at proper locations to track states. AFLNet infers program states by resolving response code from servers' response messages, which are not always available. They are either not automated or not generic. InsvCov uses heavyweight instrumentation to track values of many variables to infer invariants and estimate program state transition. StateAFL needs to compute hashes of some specific long-lived variables in the runtime to map each in-memory state as a unique protocol state. They both introduce significant overhead and reduce the efficiency of fuzzing. Therefore, a state-aware fuzzer should automatically recognize program states and track them in an efficient way.

Third, how to utilize program states to guide fuzzing? IJON replaces the code coverage bitmap used by AFL with manually-annotated state coverage. AFLNet tracks state (response code) transitions in addition to code coverage. They use one seed corpus to store both test cases of discovering new code or new states, and favor test cases that increase code coverage. It is worth exploring new feedback mechanisms to utilize program states better.

In this paper, we propose a new state-aware fuzzing approach `StateFuzz` to complement traditional code coverage guided fuzzers. `StateFuzz` utilizes critical variables to represent program states. These critical variables have the following features: they have a long lifetime; they can be updated (i.e., state transition) by users; they can affect the program's control flow or memory access pointers. We denote these critical variables as *state-variables*. The combination of all state-variables' values forms a *program state*, which is coarse-grained but useful for fuzzing.

Further, `StateFuzz` utilizes static analysis to recognize state-variables. We notice that rich-state programs (e.g., device drivers) always require multiple or multi-stage inputs. Different stages of inputs will trigger different program actions. Target programs have to track program states across program actions for synchronization and coordination, and as a result, state-variables are usually shared and accessed by different program actions. For example, the state-variables related to the login state are supposed to be shared by login request and logout request. We use static analysis to recognize program actions and state-variables from shared variables accessed by them. To efficiently track the program states, we reduce the number of state-variables used in the composition of a program state and the value space of each state-variable. First, we model a program state with *relevant* state-variable pairs rather than a combination of all state-variables. Second, for each state-variable, we recognize the set of values (or value ranges) it could take, where different value choices represent different states. And then, we divide each state-variable's value space into several ranges and track whether each range is hit during fuzzing.

Lastly, in addition to code coverage, we apply two new types of feedback and design a three-dimension feedback mechanism to guide the fuzzing process. The first type of feedback is that an input is interesting if it could hit a new value range combination of two variables and these two variables are both in a relevant state-variable pair. The second type of feedback is that an input that changes the upper or lower value bound of a state-variable so far is also interesting. This feedback still applies when the first feedback mechanism fails, i.e., when the value ranges of state-variables cannot be determined.

We have implemented a prototype of `StateFuzz` for system call-based Linux driver fuzzing, based on the fuzzing tool Syzkaller [27]. We evaluate `StateFuzz` on drivers in both the MSM-4.14 kernel used by Android Pixel-4 phones and the Linux upstream kernel v4.19. The evaluation result shows that `StateFuzz` is effective at discovering new vulnerabilities and new code. `StateFuzz` in total has discovered 2 known but unpatched vulnerabilities and 18 new vulnerabilities, among which 15 have been assigned CVE IDs or bug bounty rewards. Compared to state-of-the-art approaches, `StateFuzz` could find much more vulnerabilities and hit 19% more edges. We will release the source code of `StateFuzz` after publication[2].

In this paper, we make the following contributions:

- We propose a new fuzzing solution `StateFuzz` for rich-states programs, e.g., drivers, to promote fuzzing efficiency by incorporating program states as feedback.
- We propose to model program states with state-variables and automatically recognize states using static analysis and symbolic execution.
- We design a new three-dimension feedback mechanism to help fuzzers efficiently explore program states while increasing code coverage.
- We implemented a prototype of `StateFuzz` and evaluated it on real-world drivers, and found 18 new vulnerabilities in drivers while achieving much higher code coverage than existing approaches.

2 Background

2.1 POSIX Driver Fuzzing

In recent years, many fuzzing solutions have been proposed to find vulnerabilities, such as IMF [25] for the Mac OS kernel, iofuzz [18], ioctlfuzzer [16], ioctlbf [8], and ioattack [15] for the Windows kernel. Syzkaller [27] uses grammar-based templates to generate test cases to interact with the kernel by system call interface and utilizes KCOV [28] and KASAN [26] to track code coverage and detect memory bugs, respectively.

Everything is a file in the Linux kernel, and so are hardware devices. The POSIX standard provides a unified abstraction of hardware to user-space applications. Each file in the directory `/dev` represents a hardware device in Linux, which can be used by user-space programs just like a regular file. For example, a user-space application needs to obtain a file descriptor

[2] https://github.com/vul337/StateFuzz

of a device and then interacts with it via `read` and `write` system calls. In addition, a special system call with the prototype of `int ioctl(int fd, unsigned long request, ...)` is provided for user-space applications to support customized hardware behaviors according to the `request`.

In general, Linux drivers have two attack surfaces, one for hardware devices and the other for system calls. As a result, there are two dimensions to fuzzing Linux drivers. The first dimension is fuzzing drivers by injecting inputs from the hardware device side through configurations or I/O channels such as Port I/O, MMIO, and DMA. For example, to fuzz the probe routine of the USB drivers, USBFuzz [36] utilizes a generic USB device to match with the drivers and sends malicious USB descriptors to them. PeriScope [43] injects fuzzing data into the MMIO of the drivers via hooking page fault handlers.

The second dimension is from system calls. It is challenging to generate valid test cases since the arguments of system calls are diverse. For example, a valid *ioctl()* system call usually takes a complicated structure and a command (typically a big integer) as arguments. Syzkaller relies on human efforts to extract system call interfaces, to trigger drivers' actions. DIFUZE [14] applies static analysis to extract supported `request` types and associated arguments from customized interfaces of device drivers, which helps fuzzers to generate valid test cases.

2.2 Motivation Example

Code coverage is the most widely used feedback by fuzzers. Fuzzers get a reward signal when the test case hits new code (e.g., basic blocks, edges, or paths), and then they preserve this test case for future exploration. Coverage-guided fuzzing has been proven to be effective at exploring new code and vulnerabilities. Recent kernel fuzzing solutions or driver fuzzing solutions generally fall into this category. However, the code coverage feedback is limited for the following reasons.

First, some vulnerabilities could only happen under the premise of some prerequisite states. However, test cases that explore new code paths sometimes have a very limited contribution to exploring the program's state space. Therefore, coverage-centric fuzzers may waste computing resources on test cases useless for exploring more states and hurt the performance of finding vulnerabilities.

Second, coverage-centric fuzzers may discard test cases that trigger new states but not new paths since they do not contribute to code coverage. Therefore, coverage-centric fuzzers may miss the opportunity to find vulnerabilities under these states. For instance, if a program has rich internal states, a multi-stage input will fail to explore new code if the program state is not set correctly in the early stage. Even though a multi-stage input happens to set the program state in the early stage properly, the fuzzer would discard it because no new code is found in the early stage, and fails to use it as a starting point to yield new qualified multi-stage inputs.

```c
/* scull. c : source code file of the example driver. */
char *buf;
enum state {M0=0, M1, M2, M3, M4, M5, M6, M7};
static enum state my_state_A=0;
static uint8_t my_state_B=0;
static int scull_open(struct inode *inode, struct file *file) {
    if (!buf) {
        buf = kmalloc(0x3f, GFP_KERNEL); /* allocate memory */
        if (!buf) return -ENOMEM;
    }
    return 0;
}
static long scull_ioctl(struct file *filp, unsigned int cmd,
      unsigned long arg) {
    int retval=0, num=0;
    uint8_t ch;
    retval = copy_from_user(&ch, (uint8_t *)arg, 1);
    switch (cmd) {
        case 'V':
            if (buf && my_state_A==M3) {
                buf[my_state_B] = ch; /* OOB bug here */
                my_state_A = my_state_B = 0;
            }
            break;
        case 'A':
            num = ch - '0';
            if (num < 8 && num >= 0) my_state_A=num;
            break;
        case 'B':
            if (ch > 0x3f) return -EINVAL;
            my_state_B = ch;
            break;
        default: retval = -EINVAL; break;
    }
    return retval;
}

/* poc. c : user-space program for triggering the OOB bug. */
void poc() {
        char ch;
        int fd = open("/dev/scull", O_RDWR, 0);
        ch = '3'; ioctl(fd, 'A', &ch);
        ch = '?'; ioctl(fd, 'B', &ch);
        ioctl(fd, 'V', &ch);
}
```

Listing 1: A motivation example driver.

Listing 1 shows a motivation example. A coverage-guided fuzzer could find an input, e.g., a program containing system call `ioctl(fd,cmd='A',*arg='0')` or system call `ioctl(fd,cmd='B',*arg='b')`, to explore certain code in the fuzz target. But later, it will discard other inputs, e.g., programs with `ioctl(fd,cmd='A',*arg='3')` or `ioctl(fd,cmd='B',*arg='?')`, which do not hit new code but instead hit new program states `my_state_A=3` or `my_state_B=0x3f` (i.e., `my_state_B='?'`). As a result, it has a very small chance to yield new inputs (e.g., a program containing `ioctl(fd,cmd='A',*arg='3')` along with `ioctl(fd,cmd='B',*arg='?')`) that could trigger these two states individually or at the same time. In other words, a coverage-guided fuzzer fails to trigger the vulnerability for a long time. We have conducted a 48-hour fuzzing campaign using Szkaller for this example driver program with KASAN enabled, and we write system call templates (Appendix Listing 3) manually for the example driver. It takes Syzkaller 13 hours (14 million inputs generated) to find this vulnerability (the code coverage trend is shown in Appendix Figure 9).

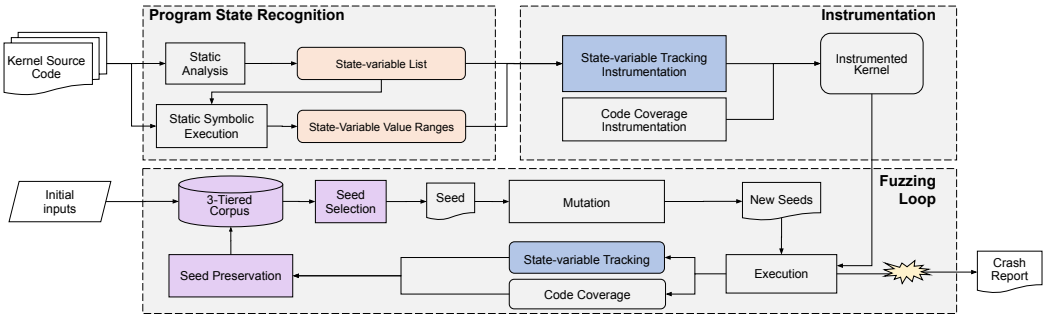


Figure 1: Overview of `StateFuzz`.

In summary, *coverage-guided fuzzers will ignore test cases that exercise the same code path, even though they have explored new states*, and have trouble discovering new code or vulnerabilities that depend on new states. Thus, it is crucial to explore program states during fuzzing and prioritize test cases that trigger new states even if they may not hit new code.

2.3 Program States

Essentially, a program state is the execution context of a program, including everything the program currently operates on, i.e., values of all program variables (from the perspective of software) and values of all virtual memory and registers (from the perspective of hardware). Exploring all potential program states will reveal all potential vulnerabilities. However, it is infeasible to track such program states during fuzzing due to computing resource limitations.

Informally, a program state is a certain execution context maintained by a program to remember preceding events or user interactions. We have conducted an empirical study on state-rich programs to learn how they represent program states. Specifically, we collected 50 code commits from open source projects containing the keyword "state machine", which indicates the program is processing certain states. The 50 commits consist of (1) all 14 Linux kernel commits in patchwork[3] with matching keywords, (2) 21 commits from March 2019 to September 2020 in the MSM kernel[4], and (3) 15 commits from popular network protocol projects in Github, including nfs-ganesha, curl, httpd, OpenSMTPD, OpenSSL and OpenSMTPD. Then, we manually analyzed how these code commits mark the program states. The result shows that, in 48 out of 50 commits, the states are represented by variables of the boolean/integer or enumeration type (Table 1 describes the sources of these 50 commits and shows several examples of variables). For the other two commits, one uses function pointers to represent states, and the other uses state code in packets to represent states.

Thus, it is very common for programs to store valuable program states into variables, and we can utilize variables that hold critical information to represent program states.

[3] https://lore.kernel.org/patchwork/

[4] https://source.codeaurora.org/quic/la/kernel/msm-4.19/

Table 1: Examples of variables that represent program states in open source projects.

Project	Example state-variables	Variable Type
Linux Kernel	agg->is_active	bool
Linux Kernel	rsclp->flags	int
nfs-ganesha	sigusr1_triggered	int
curl	conn->bits.do_more	bool
OpenSMTPD	s->state	enum
openssl	st->state	enum
httpd	session->state	enum

3 Our Solution: `StateFuzz`

To address the limitation of code coverage guided fuzzers, we propose a state-aware fuzzing solution `StateFuzz` for system call-based Linux driver fuzzing. In this section, we will present the design details of this solution.

3.1 Modeling of Program State

State-variable. We summarize the characteristics of variables used to represent program states as follows: First, these variables have a long lifetime that can span different program states to record state information. Second, they can be updated (i.e., state transition) by users. Third, since the program state always controls the program's behavior, these variables should be able to affect (directly or indirectly) the program's control flow or memory access pointers. We denote these variables as *state-variables*.

Program State. Since every state-variables could hold critical program state information, a program state ideally should contain combinations of all state-variables' values. However, the number of such combinations is too large, and it is impractical to track such states. We thus try to reduce the number of combinations with two optimizations.

First, we only consider relevant state-variable pairs in a program state, inspired by the fact that coverage-guided fuzzers in general only track edge coverage (i.e., combinations of two relevant blocks) rather than path coverage (i.e., combinations of all blocks). We conduct experiments to support this intuition, as described in Appendix A.1. We mark two state-variables as **relevant** if there are control flows or mem-

ory access pointers that are affected (directly or indirectly) by both of these two variables. For example, if two variables are checked by two conditional statements respectively, and these two conditional statements are nested, they are relevant.

Second, each state-variable's value space is large (e.g., 2^{32} for a 32-bit variable), but it only takes a limited number of values (or value ranges). Thus we propose to divide the state-variable's value space into several value ranges and track whether each value range is hit during fuzzing.

Instead of tracking their values, we track value-range combinations of two variables for every relevant state-variable pair during the fuzzing process. We denote such value-range combinations as a *value-range edge*. Furthermore, we also track extremum values of state-variables as a complement to value-range tracking. We deem that we discover *new program states* if we discover new value-range edges or new extremum values of state-variables.

3.2 Overview

Figure 1 illustrates the overview workflow of `StateFuzz`. It includes three major phases: program state recognition, program instrumentation, and the fuzzing loop.

Program state recognition. We first analyze the source code of Linux drivers to recognize program states by extracting state-variables, value choices of state-variables, and the relevance between state-variables. Specifically, we utilize static analysis to recognize program actions triggered by different stages of inputs and identify shared variables accessed by multiple program actions. To make tracking state-variables practical, we further analyze their value (or value range) choices. Then we utilize static symbolic execution to collect the value constraints of each state-variable, and infer its disjoint value-ranges.

Program instrumentation. We instrument the target program (i.e., Linux drivers) to track the program state coverage, along with code coverage (e.g., provided by KCOV [28]). Specifically, given the identified state-variables, we first perform alias analysis with SVF [44] to recognize the alias of state-variables. Then during compilation, `StateFuzz` checks the destination pointer of each store instruction. If the destination pointer points to a state-variable or a state-variable's alias, the instruction is instrumented to keep track of the value to store. More precisely, it will track value-range edges and extremum values to yield program state coverage feedback.

Fuzzing Loop. We extend the fuzzing loop of code coverage-guided fuzzers by applying the program state feedback to the process of seed input preservation and seed selection. Specifically, we will preserve inputs that discover new value-range edges or new extreme values of a state-variable, along with inputs that find new code. We then apply a delicate selection strategy to select inputs for mutation from these different types of preserved seeds. More details will be described in Section 3.4.

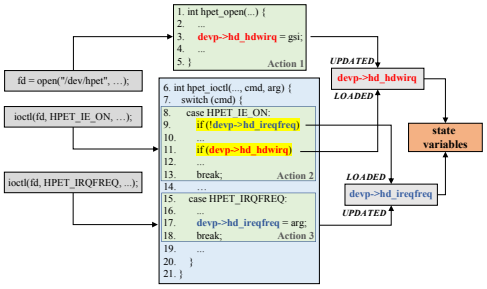


Figure 2: Example of state-variables in the `hpet` driver.

3.3 Program State Recognition

As the core of `StateFuzz`, it takes program states into consideration in addition to code coverage feedback. In this section, we will detail how to recognize program states.

Recognize Program Actions. Rich-state programs, e.g., drivers, usually require multiple or multi-stage inputs. Each stage of input will trigger specific *program actions*. For Linux device drivers, program actions are handler functions that can be invoked via system calls.

Figure 2 illustrates several example program actions, which are taken from the `hpet` driver's code. First, many program actions are initialized in global operation structures. For instance, the `hpet_open` and `hpet_read` handlers initialized in the global operation structure `hpet_fops` represent the open and read actions, which can be triggered by certain inputs. Second, the driver may take program actions via subcommands of the `ioctl` interface. For example, statements from line 8 to line 13 represent one specific program action, which could be triggered by user input of the command `HPET_IE_ON`.

Based on these conventions of Linux driver code, we utilize static analysis to recognize program actions. For system calls including read, write, open, poll and mmap, we analyze their source code to locate initializers that assign function handlers to global operation structures. These assigned function handlers are entry points of program actions. For the `ioctl` system call, we extend the tool DIFUZE [14] to perform an inter-procedural and path-sensitive analysis and recognize all supported sub-commands. The code snippet associated with each sub-command is a program action.

Recognize State-variables. Different program actions sometimes have to coordinate program states via state-variables. Thus state-variables are usually shared by multiple program actions. For example, the state-variables related to the login state are supposed to be shared by login request and logout request. To recognize the state-variables, we analyze the code of every program action based on a call graph and recognize the accessed variables. If the variables can be backtracked to global variables or fields of structures, then we will mark them as candidate *state-variables* since state-variables are supposed to have a long lifetime. After analyzing all program actions, we will filter all candidate *state-variables*. We

only keep candidate *state-variables* that will be updated by one program action and loaded by another program action (i.e., variables will be discarded if not read by any actions or not written by any actions). For instance, the variables `devp->hd_ireqfreq` and `devp->hd_hdwirq` in Figure 2 are both recognized as state-variables.

Infer State-variables' Value Ranges. To infer the value ranges of each state-variable, we perform an inter-procedural and path-sensitive static symbolic execution on the abstract syntax tree (AST) of the source code. To avoid path explosion, it is notable that our symbolic execution analyzes only a single source code file at a time (i.e., intra-module analysis), which greatly reduces the number of paths. We could locate state-variables in the AST by their type and names. Then we explore each program path and recognize path constraints related to state-variables during the exploration. We then identify two state-variables relevant if there is a path whose constraints involve both of them. By consulting with constraint solvers, we could infer critical values of the state-variable, which would change the value of path constraint. We then use these critical values as boundaries to split the value space of the state-variable.

For example, in the driver code shown in Figure 2, the state-variables `devp->hd_ireqfreq` and `devp->hd_hdwirq` are checked against 0 in the Line 9 and Line 11 respectively. The conditional statement in Line 9 leads to two branches, and the extracted constraints of these two branches are `devp->hd_ireqfreq!=0` (i.e., `devp->hd_ireqfreq<=-1` or `devp->hd_ireqfreq>=1`) and `devp->hd_ireqfreq==0`. Therefore, we can get three boundary values: -1, 0, and 1. We thus divide the value space of `devp->hd_ireqfreq` into 4 value ranges with these 3 numbers, i.e., [INT_MIN,-1],(-1,0],(0,1],(1,INT_MAX]. The case in Line 11 is similar.

3.4 Fuzzing Loop

After instrumenting the driver to track state changes, the fuzzer enters the main fuzzing loop. `StateFuzz` adopts a genetic algorithm to guide the fuzzer to explore more program states, as AFL does to increase code coverage. Specifically, `StateFuzz` preserves and prioritizes seeds that discover either new code or new states. In this section, we introduce how `StateFuzz` fine-tunes the evolutionary through a novel 3-dimension feedback mechanism, seed preservation strategy, and selection strategy.

3.4.1 Three-dimension Feedback Mechanism

`StateFuzz` adopts a novel three-dimension feedback mechanism to guide state exploration, consisting of the following three dimensions.

Code coverage dimension. `StateFuzz` reuses the code coverage feedback as existing fuzzers (e.g., Syzkaller), which issues a feedback signal whenever a test case hits new code.

Algorithm 1 Seed Preservation Algorithm

```
Input: s, the test case to be processed
Output: T1, tier 1 of corpus to store seeds discover new code edges
Output: T2, tier 2 of corpus to store seeds discover new value-range edges
Output: T3, tier 3 of corpus to store seeds discover new extreme values
  T1 = T3 = [ ], T2 = { }
  coverage, cfgPathHash ← execute(s)
  if findNewCode(coverage) then
    T1 ← T1 ∪ {s}
  end if
  if findNewValueRangeEdge(coverage) then
    if cfgPathHash IN T2 then
      T2[cfgPathHash] ← T2[cfgPathHash] ∪ {s}
    else
      T2[cfgPathHash] = ∅
      T2[cfgPathHash] ← T2[cfgPathHash] ∪ {s}
    end if
  end if
  if findNewExtremum(coverage) then
    T3 ← T3 ∪ {s}
    T3 ← minimize(T3)
  end if
```

This dimension of feedback enables the fuzzer to preserve seeds that discover new code.

Value-range dimension. `StateFuzz` applies a novel value-range dimension of feedback. If the input triggers a new value-range edge, which means triggering a new state, it issues a feedback signal. It enables the fuzzer to preserve seeds that discover new program states, and enables smart state-space exploration when working together with the genetic algorithm. For example, in the driver code shown in Figure 2, the state-variables `devp->hd_ireqfreq` and `devp->hd_hdwirq` both have 4 value ranges, i.e., [INT_MIN,-1],(-1,0],(0,1],(1,INT_MAX]. When `devp->hd_ireqfreq` changes from 0 to 1, the value range shifts from (-1, 0] to (0, 1]. Thus, it will yield a new value-range of `devp->hd_ireqfreq` and `devp->hd_hdwirq`.

Extreme value dimension. Note that sometimes we cannot resolve the value ranges of certain state-variables, due to missing constraints or unsolvable constraints. Besides, discovering a new extremum value also means that the program enters a new state. Thus, we provide another new feedback dimension that alternatively tracks state-variables' extreme values. In detail, the fuzzer records the upper and lower bounds of each state-variable in the testing history, and issues a feedback signal when a new test case sets a state-variable out of its bounds. This dimension of feedback enables the fuzzer to preserve test cases that trigger extreme values.

3.4.2 Seed Preservation and Selection Strategy

We design a three-tiered seed corpus to preserve seeds according to the type of feedback signals each seed triggers. With such a seed corpus, `StateFuzz` then periodically selects seeds from different tiers to explore new code and new states.

Seed Preservation. Given the three different feedback mechanisms, we thus provide a three-tiered seed corpus to

Algorithm 2 Seed Selection Algorithm

Input: T_1, tier 1 of corpus to store seeds discover new code edges
Input: T_2, tier 2 of corpus to store seeds discover new value-range edges
Input: T_3, tier 3 of corpus to store seeds discover new extreme values
Output: s, the selected seed

```
Pr = 3, Pc = 3                  # predefined probability hyper-parameters
r = randInt(Pr*Pc)              # generate a integer in [0, Pr*Pc) randomly
if r < Pr then
    # with probability 1/Pc
    s ← T1[ randInt() % len(T1) ]
else if r < Pr+Pc then
    # with probability 1/Pr
    keys ← getMapKeys(T2)
    pathHash ← keys[ randInt() % len(keys) ]
    bucket ← T2[pathHash]
    s ← bucket[ randInt() % len(bucket) ]
else
    # with probability 1-1/Pr-1/Pc
    s ← T3[ randInt() % len(T3) ]
end if
```

store seeds that discover new code, new value-range edges, and new extreme values, respectively. A seed can be stored in multiple tiers if it triggers multiple feedbacks at the same time. Sometimes, seeds that discover new value-range edges but execute similar code may fill up the queue, preventing the fuzzer from exploring other code. To reduce such locality, we use seeds' paths to cluster seeds and we schedule not only seeds but also clusters (i.e., buckets) in the later seed selection stage. Algorithm 1 demonstrates how `StateFuzz` preserves seeds in detail. After executing a test case, `StateFuzz` checks whether feedback signals are generated, and puts the test case into different tiers according to the feedback signals. First, if the test case finds new code, it is added to Tier-1 of the seed corpus. Second, if the test case discovers new value-range edges, `StateFuzz` adds the test case to a specific bucket in Tier-2, which is indexed by the hash of executed basic blocks' addresses. If the path of this test case is new to `StateFuzz`, `StateFuzz` creates a new bucket in Tier-2 and stores this test case, otherwise, it stores this test case in an existing bucket. Third, if the test case discovers new extreme values, it will be added to Tier-3 of the seed corpus. Then `StateFuzz` updates the record of extreme values, and removes previous seeds that discover out-of-date extreme values to minimize the corpus. Due to our minimization mechanism, the number of seeds in Tier-3 is not very large, and we do not need to use buckets to cluster seeds.

Seed Selection. Given the preserved three-tiered seed corpus, `StateFuzz` further applies a special seed selection strategy to improve the fuzzing efficiency. Algorithm 2 shows the detail of how `StateFuzz` selects seeds from the corpus. Firstly, `StateFuzz` chooses a tier of the seed corpus, according to the predefined probability hyper-parameters P_r and P_c. Here we just let the three tiers of the corpus have the same probability of being selected (i.e., P_r=3 and P_c=3), which is a naive design (we discuss how to choose the values of P_r and P_c in Appendix A.2). After tier selection, `StateFuzz` selects a seed from the chosen tier. If Tier-1 or Tier-3 is chosen, `StateFuzz` randomly selects a seed from the tier for further mutation. If Tier-2 is chosen, `StateFuzz` first randomly selects a bucket from this tier, and then selects a random seed from this bucket for further mutation. In this way, different buckets may get an equal opportunity to be selected. It avoids a local optimum case, where seeds from one larger bucket get selected more frequently than other seeds. Since each bucket represents one control flow path, this seed selection strategy will ensure different paths are explored thoroughly. Further, within each bucket, there could be multiple seeds triggering different states. This seed selection strategy will try to explore different states when this bucket is chosen.

Table 2: Implementation details of `StateFuzz`.

Component	Tool	Lines of Code
State Recognition	DIFUZE, CRIX, CSA	#2,500 (C++)
Instrumentation	LLVM Sancov, SVF	#500 (C++)
Fuzzing Loop	Syzkaller	#3,800 (Go)
Glue scripts	-	#1,000 (Python)
Total		#7,800

4 Implementation

`StateFuzz` has three major components, including program state recognition, program instrumentation, and the fuzzing loop. It also has several glue scripts. Table 2 summarizes the components and their statistics.

In the first component, `StateFuzz` uses a modified version of DIFUZE [14] to identify drivers' program actions. Besides, it recognizes state-variables shared and accessed by different program actions with an LLVM pass, in which the two-layer type-based indirect call analysis from CRIX [13] is utilized to build call graphs. It collects state-variable constraints and infers the value ranges of each state-variable via intra-module static symbolic execution through Clang Static Analyzer (CSA) [1].

In the second component, to trace state-variables more precisely, `StateFuzz` utilizes the points-to analysis tool SVF [44] to find alias of state-variables, and trace accesses to these aliases too. We mark the state-variables with names (for global variables that are not in structure type) or their types (for fields of structures) rather than the specific pointers, which is a conservative solution and requires no complex pointer analysis. And we utilize SVF to find state-variables' alias that cannot be recognized by names or types, as a complement to state-variables. All coverage and program state tracing instructions are instrumented with LLVM SanCov.

To track states, `StateFuzz` instruments target programs to trace values of state-variables. Given the state-variable list generated in Section 3.3. During compilation, `StateFuzz` instruments tracing code after each `store` instruction if its destination pointer points to a state-variable or its alias. For

operations that write to state-variables by calling memory copy functions like `copy_from_user` and `memcpy`, we parse the type of destination memory of such functions to check if state-variables are involved in destination memory according to types.

The third component is based on the existing kernel fuzzing engine Syzkaller [27]. Similar to Syzkaller, `StateFuzz` utilizes three dictionaries to store coverage for the three dimensions. For the value-range dimension, we splice the state-variable ID and the hit range ID as a value-range unit of a state-variable. Then for the two variables in a relevant state-variable pair, we compute the hash of their value-range units (i.e., the hash of two units) to represent a value-range edge. The dictionary keys are value-range edges, and values are the number of edges' hit times. For the extremum dimension, the dictionary keys are state-variable IDs and the dictionary values are their extremums.

5 Evaluation

To demonstrate the effectiveness of `StateFuzz`, we first evaluate the variable-based state model. Then, we evaluate both code coverage and state coverage of `StateFuzz`. Last and most importantly, we evaluate `StateFuzz`'s capability of discovering new vulnerabilities. We compare `StateFuzz` with two state-of-the-art system call-based Linux kernel fuzzers: Syzkaller and HFL [29]. HFL is a hybrid kernel fuzzer that infers dependencies between system calls through symbolic execution and performs very well in fuzzing the Linux driver subsystem.

In summary, we aim to answer the following questions:

- RQ1: Are the state representation expressive and meaningful? Is there any state explosion issue? (Section 5.2)
- RQ2: Can `StateFuzz` explore more states than existing approaches? (Section 5.3)
- RQ3: Can `StateFuzz` achieve higher code coverage than other existing approaches? (Section 5.4)
- RQ4: Can `StateFuzz` discover vulnerabilities in Linux drivers? (Section 5.5)
- RQ5: How do different feedback dimensions affect `StateFuzz`'s performance? (Section 5.6)

5.1 Fuzzing Evaluation Setup

We conduct fuzzing experiments for Linux drivers in two environments: Linux upstream kernel v4.19 on qemu-system-x86_64, and Qualcomm MSM-4.14 kernel on a Pixel-4, an Android phone from Google.

In the first experiment (i.e., kernel v4.19), we test the kernel running in QEMU on a *server* machine with 2 Intel Xeon CPU E5-2695 v4 (2.10GHz) and 384GB RAM running Ubuntu 16.04 LTS. For Syzkaller and `StateFuzz`, we assign each of them 8 VMs with two vCPUs per VM (i.e., 16 vCPUs assigned). For HFL, for a fair comparison, it is assigned with 4 VMS with two vCPUs per VM (i.e., 8 vCPUs assigned) and 8 additional vCPUs for symbolic execution.

In the second experiment (i.e., MSM-4.14 kernel), we test the MSM-4.14 kernel in a Pixel-4 phone rather than QEMU. Many device drivers in MSM-4.14 rely on real phone peripherals that QEMU cannot emulate, preventing the MSM kernel from booting on QEMU. As a result, HFL cannot be applied to the MSM kernel, because its symbolic engine S2E [10] relies on QEMU to perform dynamic binary translation. Instead, we conduct the phone fuzzing as follows: (1) build and flash images for the phone as instructed by the Android debug documentation[5], (2) generate and execute test cases (i.e., running `syz-fuzzer` and `syz-executor`) on the phone, and (3) monitor the fuzzer (i.e., running `syz-manager`) on a *PC* machine which is connected to the phone via USB debugging, as instructed by the Syzkaller documentation[6]. The PC machine runs Ubuntu 18.04 LTS with an Intel Core i7-8700 CPU (3.20GHz) and 32GB RAM.

In both environments, we utilize LLVM to compile the kernel with KCOV and KCOV_ENABLE_COMPARISONS enabled to collect code coverage, etc. We also enable KASAN to detect bugs. All fuzzers involved in experiments apply the same system call templates generated by DIFUZE, and only system calls extracted by DIFUZE are enabled for fuzzing. To distinguish from the original version of Syzkaller and HFL, we use Syzkaller-D and HFL-D to refer to the original Syzkaller applying DIFUZE system call templates and the original HFL applying DIFUZE system call templates, respectively. All the fuzzers involved in experiments are run with empty initial seeds. To better demonstrate the performance of fuzzers and get convergence results, we enlarge the fuzzing time budgets. We fuzz the Linux upstream kernel for 48 hours and fuzz the MSM kernel for 72 hours (since the pixel-4 device has lower computing power, we give it a bigger budget). All fuzzing time budgets do not include time spent on the state model building. To reduce bias, we repeat all experiments three times.

5.2 State Model Evaluation (RQ1)

Time cost. We conduct a pre-analysis in the aforementioned PC machine to build the state model. On average, the pre-analysis phase of `StateFuzz` takes 15 hours. In detail, state-variable recognition costs 6 hours, pointer analysis with SVF takes 2 hours, and collecting constraints by static symbolic execution costs 7 hours. We conduct intra-module symbolic execution (via Clang Static Analyzer) on each source code file for at most 1 hour. In our experiments, 43 out of 1401 files in the MSM kernel driver subsystem trigger timeout, and 117 out of 2776 files in the Linux-4.19 kernel driver subsystem trigger timeout. Note that, for a kernel under test,

[5] https://source.android.com/devices/tech/debug/kasan-kcov

[6] https://github.com/google/syzkaller/blob/master/docs/linux/setup_linux-host_android-device_arm-kernel.md

Table 3: Statistics of state-variables and their value ranges, as well as relevant state pairs.

Kernel	# Program Actions	# State-variables	# Relevant Pairs	# Value Ranges		
				Total	Avg.	Max
Linux-4.19	840	6055	25778	18921	3.12	157
MSM-4.14	1330	5037	18743	13332	2.65	193

Table 4: **State-variable classification according to variable names.** The result shows that about half of state-variables found by `StateFuzz` are classified successfully. In Linux upstream kernel v4.19, we extract 2,299 variable names and successfully category 48% among them. Specifically, of all variables, we find 4.3% in "explicit state", 14.6% in "mode" or "flag", 7.4% in "boolean", 16.8% in "size", and 4.9% in "index". The remaining 52% of variables are in the "to be determined" category. Similarly, we successfully extract 1,857 variable names in the MSM-4.14 kernel. Among these variables, we identify their categories by name for 51% of state-variables. It is interesting that the results of MSM-4.14 kernel are very close to that of Linux kernel v4.19, although MSM-4.14 kernel contains more customized Android drivers.

Kernel	Category	Amount	Percentage	Keywords in variables' names	Example
Linux-4.19	explicit state	100	4.3%	state, status	tnc_state
	mode	146	6.4%	mode, type	sel_mode
	flag	190	8.3%	flag, mask	c_cflag
	size	387	16.8%	len, size, cnt, count, num	io_lock_cnt
	index	113	4.9%	index, idx, pos, offset, /^.*id/$	done_idx
	boolean	170	7.4%	done, /^.*ed/$, /^.*ing/$, /^.*able/$, /^is_.*/$	pie_enabled
	to be determined	1193	51.9%	-	cmd_opcode, height
MSM-4.14	explicit state	100	5.4%	state, status	r_state_current
	mode	113	6.1%	mode, type	el2_mode
	flag	119	6.4%	flag, mask	logging_mask
	size	353	19.0%	len, size, cnt, count, num	client_count
	index	97	5.2%	index, idx, pos, offset, /^.*id/$	table_index
	boolean	173	9.3%	done, /^.*ed/$, /^.*ing/$, /^.*able/$, /^is_.*/$	is_mapped, is_complete
	to be determined	902	48.6%	-	hdr_hdl, dirty

the pre-analysis only introduces a one-time cost, having a negligible impact on the fuzzing process, as discussed in §7.1.

State-related Statistics. For the Linux kernel v4.19, `StateFuzz` extracts 840 program actions, shown in Table 3. After discarding pointer type variables, `StateFuzz` identifies 6,055 state-variables and 18,921 ranges. Each state-variable is split into 3 ranges on average. Variable `sk_buff.len` has the maximum number (i.e., 157) of ranges, which stores the length of a socket's data buffer and is widely used in network communications. Followed the method in Section 3.3, `StateFuzz` recognizes 25,778 relevant state-variable-pairs. Thus, on average, one state-variable may have about 4 relevant state-variables. For the MSM-4.14 kernel, 1330 program actions are identified by DIFUZE. `StateFuzz` finds 5,037 state-variables, 18,743 value ranges and 18,743 ordered relevant state-variable pairs. The variable `diag_md_session_t.peripheral_mask` has the most ranges, 193.

As the result shows, one state-variable has no more than 4 relevant state-variables on average. Although `StateFuzz` tracks value-range combinations of two variables for every *relevant state-variable pair*, the number of combinations is acceptable, i.e., 25,778 and 18,743, respectively. Further, on average, each state-variable has less than 4 (i.e., 3.12 or 2.65 respectively) value ranges. Therefore, there are less than 16 choices for each element in the program state bitmap on average. As a result, it will not cause the *state explosion* issue.

Semantics of state-variables. We classify the extracted state-variables by investigating the semantics of their names. Empirically, those variables are divided into 6 categories, which are "explicit state", "mode", "flag", "size", "index", and "boolean". First, "explicit state" contains variables with explicit words "state" in their names. Second, variables in the "mode" and "flags" categories are usually utilized to control the behaviors of the program. Third, variables in the "size" and "index" categories are often used to save the state of a shared queue or buffer. As for "boolean", those variables are often named using past tense or progressive tense of verbs to represent program states.Variables from each category contain specific keywords in their names.

We modify the Clang static analyzer to extract state-variable names from declaration statements in the AST. Then we check if those variable names contain any keywords and split them into the categories mentioned above. In detail, the keywords used are shown in Table 4. We check the variable names using keywords in the list from top to bottom. Table 4 shows the classification result. The result shows that about half of state-variables found by `StateFuzz` contain these keywords.

Table 5: Validation results of program action recognition and state-variable recognition. StateFuzz successfully recognizes 99% of program actions and 90% of state-variables. Compared to the approach that checks whether their names contain state-related keywords, StateFuzz can recognize 49% more state-variables.

	Kernel	Driver	# Program Action			# State-variable (StateFuzz)			# State-variable (Keyword-match)			# Declared Variable
			TP	FP	FN	TP	FP	FN	TP	FP	FN	
1	Linux-4.19	OSS Sequencer	64	0	0	45	46	1	18	15	28	2673
2		PPP	26	1	0	29	44	9	25	48	13	6430
3		TUN	35	0	0	29	93	0	25	22	4	1700
4		UINPUT	24	0	2	15	24	1	9	2	7	388
5	MSM-4.14	Ashmem	15	0	0	8	30	1	6	4	3	173
6		IAxxx Cell	49	0	0	17	13	0	7	47	10	227
7		NPU	13	0	0	30	42	8	26	76	12	2528
8		SMCInvoke	6	0	0	24	7	4	16	10	12	332
	Total		**232**	**1**	**2**	**197**	**299**	**24**	132	224	89	14421

False positives and false negatives. To evaluate the accuracy of our static analysis, we randomly select 4 drivers for verification from Linux-4.19 and MSM-4.14, respectively.

Accuracy of program action recognition. We manually identify all program actions for these drivers to construct a ground truth and then validate the program actions recognized by StateFuzz. As shown in Table 5, among all 234 actions of the 8 drivers, StateFuzz successfully recognizes 99% of actions with only 1 false positive and 2 false negatives. The false positive is caused by DIFUZE treating a condition statement as an ioctl command check. On the other hand, DIFUZE misses two sub-commands when recognizing ioctl commands, resulting in 2 false negatives.

Accuracy of state-variable recognition. We evaluate the state-variables recognized by StateFuzz. Unfortunately, it is not feasible to manually validate all variables and identify state-variables in these drivers, because there could be thousands of variables declared in a driver (e.g., there are 2673 variables in the OSS sequencer driver according to our AST analysis). Instead, we only collect the variables whose names contain the keywords mentioned above and the candidate state-variables recognized by StateFuzz, and manually verify these collected variables to construct an approximate ground truth. As a result, 659 variables are collected, of which 303 are collected by StateFuzz, 163 by the keyword-matching method, and 193 by both. We manually verify all of these 659 variables and identify 221 final state-variables according to our definition of state-variables in Section 3.1. StateFuzz successfully recognizes 197 state-variables, accounting for 90% of the total, 49% more than the keyword-matching method.

The false positives are introduced for three reasons: first, a driver usually communicates with other parts of the kernel (e.g., file system, network) by reading and writing variables like inode.i_size declared outside the driver. StateFuzz may wrongly mark these variables as state-variables though they do not represent states of the driver. Second, since StateFuzz traverses instructions and collects state-variables based on the call graph, incorrect callees in the call graph could lead to false positives. Third, some candidate state-variables are only utilized for debugging, output, or sending back to userspace. These state-variables do not affect the control flow and data flow of the driver. Of the 299 false positives produced by StateFuzz, 141 are introduced by the first reason, 13 by the second, and 145 by the third.

We have tried to mitigate false positives introduced by the first reason. Specifically, we first try to recognize common utility functions that are shared by different drivers and the kernel, and then remove their instructions from the execution traces of program actions. As a result, variables only accessed in common utility functions will not be marked as state-variables. We heuristically mark functions that are called by more than MAX_NUM functions as utility functions. To prevent side effects like false negatives, we conservatively set the heuristic threshold MAX_NUM to 300, since core driver functions are unlikely to get called by more than 300 functions.

We also further investigate the impact of these false-positive state-variables on StateFuzz's fuzzing campaigns for Linux-4.19 and MSM-4.14. We find that 54 (18%) false-positive state-variables are never accessed throughout a fuzzing campaign, so no fuzzing inputs are preserved in StateFuzz's corpus for these variables. Other 155 (52%) variables have no inferred value ranges, so these variables can not introduce fuzzing inputs to the corpus for discovering new value-range edges. As a result, 209 of 299 (i.e., 70%) false positives introduce a negligible impact on the fuzzing campaigns. Overall, the effect of false positives in state-variable recognition is acceptable in the fuzzing campaigns.

The main reason for false negatives are summarised as follows: First, when building LLVM bitcode files, related functions are not linked if they are located in different modules, resulting in lacking analysis of the load and store instructions inside the functions. Second, the lack of target callees of indirect calls in the call graph can also lead to false negatives. Third, the state-variables read or written through wrappers are ignored by StateFuzz. For example, the atomic_inc function is often used to update reference counts. Note that, we did not find false negatives introduced by our aforementioned

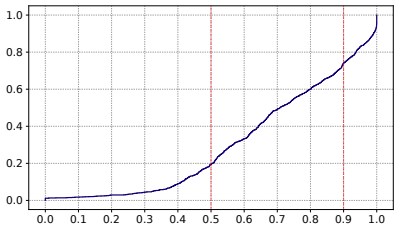


Figure 3: CDF curve of X/(X+Y). X is the count of an individual value range hit by StateFuzz, and Y is the count of an individual value range hit by Syzkaller-D. For more than 80% of ranges, (X/X+Y) is greater than 1/2, meaning that StateFuzz hits the range more frequently than Syzkaller-D.

Table 6: The number of value-range edges found by StateFuzz and Syzkaller-D-col.

Kernel	StateFuzz	Syzkaller-D-col
Linux-4.19	27117 (132%)	20507 (100%)

utility function filtering strategy, showing the conservative threshold for MAX_NUM is reasonable.

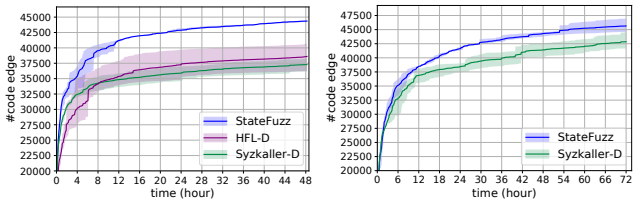
In summary, StateFuzz can recognize 99% of program actions and identify about 90% of state-variables, demonstrating the effectiveness of our static analysis.

5.3 State Coverage Evaluation (RQ2)

To demonstrate the capability of StateFuzz at exploring program states, we first compare it with a modified version of Syzkaller-D, in which we only apply additional state-variable tracking instrumentation to collect values of state-variables. We note it as Syzkaller-D-col. We do not introduce any new feedback mechanism in Syzkaller-D-col, and the instrumentation is only utilized for logging. We perform this experiment on the Linux upstream kernel of the 4.19.113 version.

Hit Count of Value Ranges. We then collect hit times of each individual value range for both StateFuzz and Syzkaller-D-col. For better demonstration, we compute the ratio of two fuzzers' accessing times for every value range R. Suppose R is accessed X times by StateFuzz and Y times by Syzkaller-D-col. If X/(X+Y) > 0.5, it means that StateFuzz accesses the range R more often than Syzkaller-D-col. Therefore, we sort values of X/(X+Y) for all state-variable ranges. The result can be presented with a cumulative distribution function (CDF) curve in Figure 3. It shows that, more than 80% of ranges' ratio exceeds 0.5, which means that StateFuzz accesses more times than Syzkaller-D-col for 80% of ranges. Besides, about 20% of ranges' ratio is greater than 0.9, which shows that StateFuzz accesses these ranges 9 times more than Syzkaller-D-col. The result shows that StateFuzz can explore states that are rarely accessed.

Hit Count of Value-range Edges. Table 6 shows the number of distinct value-range edges found by StateFuzz and Syzkaller-D-col. In the Linux upstream 4.19 kernel, Syzkaller-D-col discovers 20,701 value-range edges while StateFuzz discovers 27,117 value-range edges, 32% more than Syzkaller-D-col. As shown above, StateFuzz can explore not only value ranges that are rarely accessed but also more value-range edges, which means StateFuzz can explore more states with the guidance of our state model.



(a) Linux-4.19 (b) MSM-4.14

Figure 4: Code coverage. The first 20,000 code edges are hidden, as all the fuzzers discover them in 10 minutes.

5.4 Code Coverage Evaluation (RQ3)

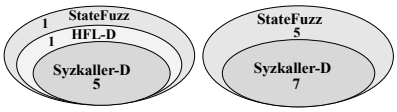
To verify whether StateFuzz is capable of exploring more code in the same time budget, we compare it with HFL and Syzkaller. This experiment is performed in both Linux kernel v4.19 and MSM-4.14 kernel. Since HFL does not support fuzzing real Android devices, we only use HFL-D to fuzz the Linux upstream kernel with QEMU.

As Figure 4 shows, our approach StateFuzz shows an advantage of code edge coverage. In Linux-4.19, StateFuzz discovers 19% more code edges than Syzkaller-D and 15% more code edges than HFL-D, while in MSM-4.14 StateFuzz discovers 7% more code edges than Syzkaller-D. The result shows that StateFuzz can achieve higher code coverage than the state-of-the-art kernel fuzzers in the same time budget.

5.5 Vulnerability Discovery Evaluation (RQ4)

We intermittently fuzzed the Linux-4.19 kernel and the MSM-4.14 kernel with StateFuzz over two months. In summary, a total of 20 vulnerabilities are found by StateFuzz, of which 7 are found in Linux-4.19 and 13 are found in MSM-4.14. All vulnerable drivers found in MSM-4.14 rely on Qualcomm SoCs or specific peripherals of Google Pixel phones, and their code is not included in the Linux upstream kernel. We reported all of these 20 vulnerabilities to developers, and 19 of them are confirmed. Table 7 shows the vulnerabilities found by StateFuzz. For security concerns, we hide function names and file names for vulnerabilities that are not fixed yet. Among the 19 confirmed vulnerabilities, 14 are assigned CVE IDs, 3 are in a pending state, and the other 2 have been discovered by developers internally[7]. Specifically, 9 confirmed vulnerabilities have been assigned bug bounty rewards by Google or Qualcomm.

[7]But patches have not been applied to the latest open-source code when we submitted reports.



(a) Linux-4.19 (b) MSM-4.14

Figure 5: The number of vulnerabilities discovered in 3 fuzzing campaigns for the Linux upstream kernel and the MSM kernel (HFL-D is only used in the Linux kernel fuzzing campaign).

To further demonstrate StateFuzz's efficiency of vulnerability discovery, we use Syzkaller-D, HFL-D, and StateFuzz to fuzz our targets, and compare the number of vulnerabilities discovered by the three fuzzers. Since HFL does not support fuzzing real Android devices, we only use HFL-D for Linux-4.19. We repeat each fuzzing campaign three times and accumulate the count of vulnerabilities together. We use the same time budgets as mentioned in Section 5.1. The result is shown in Figure 5. In this experiment, StateFuzz discovers 19 of 20 reported vulnerabilities, 46% more than Syzkaller-D. Specifically, StateFuzz discovers all 7 vulnerabilities in Linux-4.19, where HFL-D discovers 6, and StateFuzz discovers 12 vulnerabilities in the MSM kernel. The only missing vulnerability (the out-of-bounds writing in MSM diagnostic driver) is also not found by Syzkaller-D in this experiment. Besides, StateFuzz discovers all 13 vulnerabilities found by Syzkaller-D and HFL-D in this experiment, showing that StateFuzz is effective at discovering vulnerabilities.

We further investigate the 5 vulnerabilities only discovered by StateFuzz in MSM-4.14. Out of these 5 vulnerabilities, 4 vulnerable code pieces are discovered by both StateFuzz and Syzkaller-D. It demonstrates that StateFuzz can better discover program states to trigger vulnerabilities after discovering vulnerable code. HFL relies on the emulator and fails to test MSM-4.14 on the phone.

5.6 How do different feedback dimensions affect StateFuzz's performance? (RQ5)

To understand how StateFuzz works and evaluate the contribution of each feedback dimension, we compile three variants of StateFuzz by disabling each feature. Then we perform fuzzing campaigns in the Linux-4.19 kernel with variants. We only enable the code coverage feedback to implement a baseline variant named "C". The variant C-R enables the feedback dimension of value-range tracking and code coverage. Another variant C-E enables the feedback dimension of extremum tracking along with code coverage, while our complete approach C-R-E enables all three feedback dimensions.

Figure 6 demonstrates both extremum tracking and value-range tracking can contribute to code coverage. C-E achieves 9% higher code edge coverage than baseline C, and C-R achieves 10% higher. C-R-E achieves the highest growth of 17%, which means that the three feedback dimensions

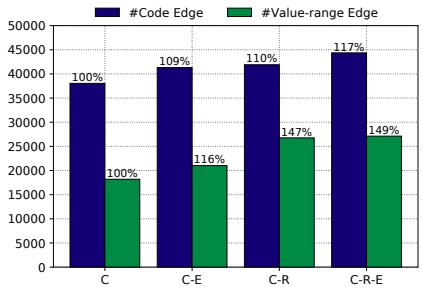


Figure 6: The number of discovered code edges and value-range edges for the Linux-4.19 kernel. C is the baseline with enabling only code coverage; C-E enables the extremum dimension along with code coverage; C-R enables the value-range dimension and code coverage; C-R-E indicates the variant that enables all three feedback dimensions.

```
1  // global variable, controlled by user through ioctl TIOCLINUX
2  static int s_delta;
3  // the parameter "lines" is equal to s_delta
4  static void vgacon_scrolldelta(struct vc_data *c, int lines) {
5    int start, end, count, soff;
6    ...
7    v_cur->restore = 0;
8    start = v_cur->cur + lines;
9    ...
10   if (start > v_cur->cnt)
11     start = v_cur->cnt;
12   ...
13   if (end > v_cur->cnt)
14     end = v_cur->cnt;
15   v_cur->cur = start;
16   count = end - start;
17   soff = v_cur->tail - ((v_cur->cnt - end) * c->vc_size_row);
18   soff -= count * c->vc_size_row;
19   if (soff < 0)
20     soff += v_cur->size;
21   ...
22   // out of bounds read when soff < 0
23   memcpy(d, v_cur->data + soff, copysize);
```

Listing 2: A simplified code snippet of CVE-2020-28097

could promote each other in discovering more code. Figure 6 also demonstrates that C-R discovers 47% more value-range edges than baseline C, while C-E discovers only 16% more. Compared to C-R, the value-range edge coverage of C-R-E barely increases even with enabling additional extremum tracking. This experiment shows that value-range tracking contributes the most to increasing value-range edge coverage.

5.7 Case Study: CVE-2020-28097

Listing 2 demonstrates a simplified out-of-bound memory read vulnerability in the VGA console driver, which is only discovered by StateFuzz. The variable s_delta is a global variable which represents the scroll state of vga console. s_delta and struct v_cur can be controlled by user through ioctl, and the value of s_delta will be copied to the parameter lines of function vgacon_scrolldelta. The root cause of this vulnerability is that the offset soff of memcpy's source buffer can be negative, leading to an out-of-bounds memory

Table 7: Vulnerabilities found by StateFuzz. Status *Confirmed*^{B} indicates that the vulnerability has been confirmed and assigned a bug bounty reward by the vendor. Status *Confirmed** means that the vulnerability has been found by developers internally, but the patch was not released or merged when we reported the vulnerability. Function and file names of non-fixed vulnerabilities are hidden for security concerns.

	Kernel	File	Function	Vulnerability Type	Status	CVE ID
1		drivers/input/keyboard/sunkbd.c	sunkbd_reinit	Use-after-free	Confirmed & Fixed	CVE-2020-25669
2		drivers/staging/speakup/spk_ttyio.c	spk_ttyio_ldisc_close	Null-pointer Dereference	Confirmed & Fixed	CVE-2020-28941
3		drivers/staging/speakup/spk_ttyio.c	spk_ttyio_receive_buf2	Null-pointer Dereference	Confirmed & Fixed	CVE-2020-27830
4	Linux-4.19	drivers/video/console/vgacon.c	vgacon_scrolldelta	Out-of-bounds Read	Confirmed & Fixed	CVE-2020-28097
5		drivers/md/dm-ioctl.c	list_devices	Out-of-bounds Write	Confirmed & Fixed	CVE-2021-31916
6		drivers/bluetooth/		Use-after-free	Reported	
7		drivers/tty/vt/		Deadlock	Confirmed	
8		drivers/mfd/adnc/iaxxx-module.c	iaxxx_core_sensor_change_state	Out-of-bounds Read	*Confirmed*^{B} & Fixed	CVE-2021-0461
9		drivers/platform/msm/ipa/ipa_v3/ipa_utils.c	ipa3_counter_remove_hdl	Out-of-bounds Read	Confirmed & Fixed	CVE-2021-30265
10		drivers/char/diag/diag_pcie.c	diag_pcie_write	Out-of-bounds Write	*Confirmed*^{B} & Fixed	CVE-2021-30298
11		drivers/char/diag/diag_dci.c	diag_send_dci_pkt_remote	Out-of-bounds Write	*Confirmed*^{B} & Fixed	CVE-2021-30324
12		drivers/char/diag/diag_dci.c	extract_dci_pkt_rsp	Out-of-bounds Write	*Confirmed*^{B} & Fixed	CVE-2021-30325
13		drivers/mfd/adnc/iaxxx-btp.c	iaxxx_btp_write_words	Out-of-bounds Read	*Confirmed*^{B} & Fixed	CVE-2021-39717
14	MSM-4.14	drivers/misc/faceauth_hypx.c	hypx_create_blob_dmabuf	Use-after-free	*Confirmed*^{B} & Fixed	CVE-2022-20183
15		drivers/misc/ipu/ipu-core-jqs-msg-transport.c	ipu_core_jqs_msg_transport_kernel_write_sync	Use-after-free	*Confirmed*^{B} & Fixed	CVE-2022-20155
16		drivers/mfd/abc-pcie.c	abc_pcie_enter_el2_handler	Use-after-free	*Confirmed*^{B} & Fixed	CVE-2022-20185
17		drivers/nfc/		Use-after-free	*Confirmed*^{B}	
18		drivers/char/diag/		Out-of-bounds Read	Confirmed	
19		drivers/platform/msm/ipa/ipa_v3/ipa_odl.c	ipa3_replenish_rx_cache	User-after-free	*Confirmed** & Fixed	
20		drivers/char/adsprpc.c	get_args	Null-pointer Dereference	*Confirmed** & Fixed	

read from a negative offset at Line 23. When the value of s_delta is set to a small negative number and v_cur->cnt is set to a large positive number, soff will be a negative integer after a series of computations.

We observe that Syzkaller-D can easily cover all the relevant code with the guidance of code coverage feedback. But Syzkaller-D fails to find a proper test case to trigger this vulnerability, since the probability of setting these two variables to proper values is quite low. However, StateFuzz successfully identifies the variable s_delta as a state-variable, and preserves the seed which hits the negative value range of s_delta for future mutation. As a result, StateFuzz has more chances to execute the vulnerable code with a program state in which s_delta is negative and thus can trigger this vulnerability much easier. In all 3 of our 48-hour experiments, according to our log, on average StateFuzz enters this special state (i.e., s_delta is negative) for 5400 times, while Syzkaller-D only enters this state for 16 times. It implies that StateFuzz can guide fuzzer to explore different program states and discover potential vulnerabilities.

6 Related Work

State Model Building State models have been widely studied in research about network protocols. Recent work builds the protocol state model based on network traffic or dynamic taint analysis [13, 23, 24]. In addition, recent works also utilized state protocol fuzzing to infer TLS/DTLS protocol state models for verifying whether the implementation of a protocol is secure [7, 11, 17, 21, 42]. Ferry [54] dynamically recognizes those variables that determine condition branches and are influenced by inputs as state-describing variables. Compared to StateFuzz, Ferry focuses on exploring state-related code branches, while StateFuzz tries to explore more value ranges of state-related variables during fuzzing. Besides, these methods based on dynamic analysis are limited by completeness.

State-aware Fuzzing Memlock [48] uses the extremum value of the program's allocated memory size as an additional feedback dimension for grey-box fuzzing to discover memory leaks. But Memlock ignores the non-extremum value ranges and can not track fine-grained states inside programs. IJON [4] proposed an annotation mechanism involving human analysis to guide the fuzzer to learn the program's internal state and track specific variables. However, since annotation requires prior knowledge about the target program, it consumes additional manual efforts. Instead, InvsCov [19] tracks all variables if they are involved in memory access instructions or if they represent return values. Then InvsCov infers immutable boundaries for tracked variables by detecting invariants from traces of pre-acquired corpus, and send additional feedback when tracked variables violate these boundaries. StateFuzz recognizes state-variables more accurately by using static analysis to mine programs' semantic information. StateFuzz does not need any corpus, and StateFuzz can update the boundaries via our extremum feedback mechanism as the exploration advances.

In state-aware protocol fuzzing, AFLNet [38] and SGPFuzz [52] identified states based on response codes in network packets to achieve state-aware fuzzing. RESTler [5] infers execution states of inputs according to the response code of REST API. However, response code is not common and not available in most scenarios. StateAFL [32] proposes to use locality-sensitive hashing on runtime memory to represent program states in protocol fuzzing. It utilizes expensive post-execution analysis to maintain the map between memory and states during fuzzing, thus reducing the fuzzing efficiency. KiF [2], SNOOZE [6] and these open-source tools [3, 35, 37] require prior knowledge of the protocol state model to per-

form stateful fuzzing, leading to a lack of scalability. Mobile device fuzzer Vulcan [50] needs to collect application logs to build an explicit state machine. StateFuzz can track program states in a lightweight way and does not rely on response codes or prior knowledge.

Linux kernel and driver fuzzing Syzkaller [27] is an engine deployed by google for fuzzing kernel with system calls. Developers manually added a bunch of system call templates for Linux drivers. DIFUZE [14] focuses on automatically extracting ioctl entry points, corresponding structures, and device names for Linux driver fuzzing. Ex-vivo [39] extracts ioctl entries and implements a kernel to fuzz Android drivers without real devices. Periscope [43] and USBFuzz [36] are fuzzers aiming to fuzz Linux drivers by executing test cases from the hardware side. Unlike StateFuzz, Periscope and USBFuzz inject inputs through configuration files or multiple I/O channels such as MMIO and DMA, rather than system calls. Charm [45] proposed a system solution for running device drivers of mobile systems in a virtual machine to enable existing analysis solutions, such as fuzzing. NTFuzz [12] performs static binary analysis to infer system call types for system call-based Windows kernel fuzzing. IMF [25] aims to infer explicit input dependencies between system calls by tracking system call traces on MacOS. MoonShine [34] focuses on retrieving the dependencies of system calls by statically analyzing the parameters and accessing global variables. SyzVegas [46] dynamically and automatically adapts Syzkaller's task scheduling along with seed selection to improve code coverage by leveraging multi-armed-bandit algorithms and a novel reward assessment model. HFL [29] is a hybrid kernel fuzzer based on Syzkaller, and HFL infers dependencies of system calls with dynamic symbolic execution. However, they all lack further analysis of program states Krace [49] utilized a two-dimension coverage mechanism to fuzz the Linux file system for concurrency bugs.

7 Discussion

7.1 Performance Impact of Pre-analysis

To better guide fuzzing, it is a common practice to perform pre-analysis to extract useful knowledge before fuzzing [9, 14, 33, 34, 47]. Although the pre-analysis (i.e., state model building by StateFuzz) introduces time cost, we believe the performance impact to fuzzing is negligible for two reasons. First, StateFuzz only needs to perform pre-analysis once for one target kernel, and the results are saved as files and can be reused by further fuzzing. Second, fuzzing campaigns usually last for days or even months, especially in kernel-space fuzzing. As fuzzing campaigns last longer, the computing resource spent on pre-analysis becomes less significant. In summary, we believe that such a pre-analysis is meaningful for improving fuzzing and the overhead is acceptable.

7.2 Effects of Static Analysis Accuracy

StateFuzz's performance can be affected by the accuracy of the static analysis. Specifically, program action recognition and state-variable recognition are key factors in conducting state-aware fuzzing, and program action recognition dominates the range of the state-variable recognition. False negatives of state-variables lead to poor feedback about program states during fuzzing, thus reducing the performance contributed by the value-range dimension and the extremum dimension. False positives of state-variables cause many useless test cases to fill up the seed corpus, wasting computing resources. Since we mark state-variables with types or names rather than specific pointers, we do not differ pointers with the same types. Such a conservative way benefits two aspects. First, it avoids many false negatives caused by imprecise pointer analysis. Second, value-ranges from different source code files are merged according to state-variables' types or names, avoiding the false negatives caused by intra-module analysis's limited range. Similar to state-variable recognition, false positives caused by intra-module analysis and alias analysis also lead to many useless test cases and a waste of computing resources.

7.3 Limitations and Future Work

Soundness and Completeness. Thanks to the fast execution, fuzzing can tolerate some false positives caused by the program state recognition. We believe that the completeness is more worthy of consideration since the number of StateFuzz's recognized state-variables is insignificant. In the future, we will work to address the issues of incomplete linkage and wrapper function identification.

Fuzzing other targets. The prototype of StateFuzz mostly focuses on system call-based Linux driver fuzzing. On the one hand, Linux drivers are stateful, and most of them interact with user space in the same manner (i.e., through system calls). Besides, the fuzzing framework Syzkaller can be utilized to quickly implement a prototype of our approach. On the other hand, for targets like network protocols, few handy fuzzing frameworks can provide feedback transmission and synchronize stages between client and server. We believe the effort of building such a framework is orthogonal to our work.

Some Linux drivers (such as USB) could interact with users through multiple I/O channels rather than system calls. StateFuzz should be extended to recognize the program actions, which do not follow conventions of system calls. Specifically, we can trace the value-flow of inputs by lightweight instrumentation to dynamically find the entry functions that handle our inputs.

StateFuzz is applicable to fuzz programs supporting sequential interaction events (e.g., network protocols, system calls, GUI message loops, event loops, smart contracts in the blockchain). For instance, vulnerabilities in smart contracts in general rely on a sequence of interactions to trigger. Each invo-

cation to the smart contract will change certain *states*, which eventually leads the contract to a vulnerable state. StateFuzz could recognize such critical states and interfaces that can alter the states, which guides the fuzzer to efficiently discover vulnerabilities in them.

8 Conclusion

In this paper, we assessed the limitation of coverage-guided fuzzing solutions, and proposed a state-aware fuzzing solution StateFuzz. It utilizes static analysis to recognize shared variables that are accessed by multiple program actions, and use them as state-variables to characterize program states. By tracing values of state-variables and using a combination of two state-variables as feedback, StateFuzz can explore states efficiently during fuzzing while increasing code coverage. We implemented a prototype of StateFuzz for Linux and Android driver testing. It has discovered 20 vulnerabilities in Linux upstream drivers and Android drivers. Moreover, StateFuzz can achieve higher code coverage and state coverage than existing driver fuzzing approaches.

Acknowledgements

We would like to sincerely thank all the anonymous reviewers and our shepherd, Dr. Suman Jana, for their valuable feedback that greatly helped us to improve this paper. This work was supported by the National Key Research and Development Program of China (2021YFB2701000), National Natural Science Foundation of China (U20B2046, 61972224), Beijing National Research Center for Information Science and Technology (BNRist) under Grant BNR2022RC01006.

References

[1] Clang static analyzer. https://clang-analyzer.llvm.org/.

[2] Humberto J Abdelnur, Radu State, and Olivier Festor. Kif: a stateful sip fuzzer. In *Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, pages 47–56, 2007.

[3] P. Amini. A pure-python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley.

[4] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612. IEEE, 2020.

[5] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE, 2019.

[6] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. Snooze: toward a stateful network protocol fuzzer. In *International Conference on Information Security*, pages 343–358. Springer, 2006.

[7] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552. IEEE, 2015.

[8] J. brun. Windows kernel drivers fuzzer. https://github.com/koutto/ioctlbf.

[9] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1580–1596. IEEE, 2020.

[10] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 265–278. ACM, 2011.

[11] Chia Yuan Cho, Domagoj Babi ć, Eui Chul Richard Shin, and Dawn Song. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 426–439, 2010.

[12] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 677–693. IEEE, 2021.

[13] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *2009 30th IEEE Symposium on Security and Privacy*, pages 110–125. IEEE, 2009.

[14] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[15] Microsoft Corporation. How to perform fuzz tests with iospy and ioattack. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/how-to-perform-fuzz-tests-with-iospy-and-ioattack.

[16] Cr4sh. Ioctl fuzzer - windows kernel drivers fuzzer. https://github.com/Cr4sh/ioctlfuzzer, 2011.

[17] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of {TLS} implementations. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 193–206, 2015.

[18] debasishm89. A mutation based user mode (ring3) dumb in-memory kernel driver (ioctl) fuzzer/logger. https://github.com/debasishm89/iofuzz, 2014.

[19] Andrea Fioraldi. Program state abstraction for feedback-driven fuzz testing using likely invariants. *arXiv preprint arXiv:2012.11182*, 2020.

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[21] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of {DTLS} implementations using protocol state fuzzing. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[22] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[23] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*, pages 330–347. Springer, 2015.

[24] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *IJCSNS*, 10(8):239, 2010.

[25] HyungSeok Han and Sang Kil Cha. Inferred model-based fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[26] Google Inc. Kerneladdresssanitizer (kasan). https://github.com/google/kasan.

[27] Google Inc. syzkaller - kernel fuzzer. https://github.com/google/syzkaller.

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

[29] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Insik Shin, Yeongjin Jang, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kerne. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2020.

[30] Caroline Lemieux and Koushik Sen. Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *the 33rd ACM/IEEE International Conference*, 2018.

[31] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1949–1966, 2019.

[32] Roberto Natella. Stateafl: Greybox fuzzing for stateful network servers. *arXiv preprint arXiv:2110.06253*, 2021.

[33] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. {ParmeSan}: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306, 2020.

[34] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 729–743, 2018.

[35] PeachTech. Peach fuzzer. https://www.peach.tech/.

[36] Hui Peng and Mathias Payer. Usbfuzz: A framework for fuzzing {USB} drivers by device emulation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2559–2575, 2020.

[37] J. Pereyda. boofuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz.

[38] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *Proc. IEEE International Conference on Software Testing, Verification and Validation (Testing Tools Track)*, 2020.

[39] Ivan Pustogarov, Qian Wu, and David Lie. Ex-vivo dynamic analysis framework for android device drivers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1088–1105. IEEE, 2020.

[40] Kostya Serebryany. Libfuzzer: A library for coverage-guided fuzz testing (within llvm).

[41] Kostya Serebryany. OSS-Fuzz - google's continuous fuzzing service for open source software. In *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, August 2017. USENIX Association.

[42] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D Keromytis, and Suman Jana. Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 521–538. IEEE, 2017.

[43] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*, 2019.

[44] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[45] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 291–307, 2018.

[46] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. {SyzVegas}: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758, 2021.

[47] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 999–1010. IEEE, 2020.

[48] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *42nd International Conference on Software Engineering*. ACM, 2020.

[49] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660. IEEE, 2020.

[50] Edgardo Barsallo Yi, Heng Zhang, Amiya K Maji, Kefan Xu, and Saurabh Bagchi. Vulcan: Lessons on reliability of wearables through state-aware fuzzing. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 391–403, 2020.

[51] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 769–786. IEEE, 2019.

[52] Yingchao Yu, Zuoning Chen, Shuitao Gan, and Xiaofeng Wang. Sgpfuzzer: A state-driven smart graybox protocol fuzzer for network protocol implementations. *IEEE Access*, 8:198668–198678, 2020.

[53] Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2014.

[54] Shunfan Zhou, Zhemin Yang, Dan Qiao, Peng Liu, Min Yang, Zhe Wang, and Chenggang Wu. Ferry: State-aware symbolic execution for exploring state-dependent program paths.

A Appendix

A.1 Design choices of state-variable pairs

Like code-coverage-guided fuzzing, we mainly consider the trade-off between sensitivity and size of the seed corpus to choose what combination to track. Specifically, if we track the combination of all state-variables, due to too many seeds being preserved, maintaining a seed corpus can be expensive, and most seeds have a slight chance of being selected. On the other hand, tracking only one state-variable would miss many potential states for the coarse-grained feedback. We conduct a 48-hour fuzzing experiment in the Linux-4.19 kernel to evaluate the performance of `StateFuzz` with a different granularity of value-range feedback. Table 8 demonstrates that the fuzzing process achieves the highest code coverage and finds the most value ranges when tracking combinations of **two** relevant state-variables (i.e., state-variable pairs). The result also shows that the number of seeds in the corpus grows rapidly as the feedback granularity becomes finer. It is not feasible to track combinations of all state-variables due to corpus explosion.

A.2 Values of P_r and P_c

To evaluate the effects of different P_r and P_c utilized in Algorithm 2, we conduct an extra 48-hour fuzzing experiment in the Linux-4.19 kernel. First, we consider the situation where

Table 8: Results of fuzzing with tracking different combination types of relevant state-variables.

Combination	# Code Edge	# Value Range	# Seed in Corpus
Monuple	41563	3282	6339
Pair	**44359**	**3439**	**10808**
Triple	41793	3252	29075
Quadruple	32450	2799	38604

P_r	#Code Edge (σ=1.2%)				#Value-range Edge (σ=1.6%)			
	P_c=2	P_c=3	P_c=4	P_c=5	P_c=2	P_c=3	P_c=4	P_c=5
2	99.1%	97.2%	101.5%	97.9%	99.2%	97.7%	99.1%	98.9%
3	98.1%	100.0%	99.5%	97.8%	96.8%	100.0%	98.4%	98.6%
4	99.8%	97.0%	98.6%	99.5%	96.5%	95.5%	98.6%	98.1%
5	98.8%	99.2%	96.8%	99.5%	99.5%	96.4%	94.1%	97.4%

Figure 7: The number of code edges and value-range edges discovered by `StateFuzz` in the Linux-4.19 kernel when P_r and P_c traverse the range from 2 to 5. To better demonstrate the degree of change, we convert all numbers to percentages of the case where P_r=3 and P_c=3. The standard deviation of code edges is 1.2% and of value-range edges is 1.6%, which means that the performance of `StateFuzz` is close when using these configurations.

the first two tiers of the corpus are selected with similar high probabilities. We set P_r and P_c to be the integers ranging from 2 to 5, respectively. As shown in Figure 7, the performance of `StateFuzz` is close in these cases.

Second, we make the first two tiers of the corpus have a much smaller probability of being selected (e.g., 1/100), respectively. The result in Figure 8 shows the performance of `StateFuzz` declines significantly compared to the case where P_r and P_c are integers ranging from 2 to 5. Based on the above results, we naively apply the configuration of P_r=3 and P_c=3, with which `StateFuzz` performs well in both discovering code edges and value-range edges. As for how to find the optimal values for P_r and P_c (i.e., how to assign energy to the three tiers of corpus), we believe this is an interesting topic worthy of further study, and we leave it to future work.

A.3 Fuzzing the motivation example driver

We fuzz the motivation example driver mentioned in Section 2.2 with both Syzkaller and `StateFuzz`. In these fuzzing campaigns, we replace all system call templates of the fuzzers with our manually written templates shown in Listing 3, to help the fuzzers dispatch all program actions of the example driver. The code coverage trend of Syzkaller is shown in Figure 9). It takes Syzkaller 13 hours to find the first crash. And it still takes Syzkaller 4 hours to trigger the crash after discovering all relevant code. However, `StateFuzz` triggers the first crash in less than 5 minutes in our experiments.

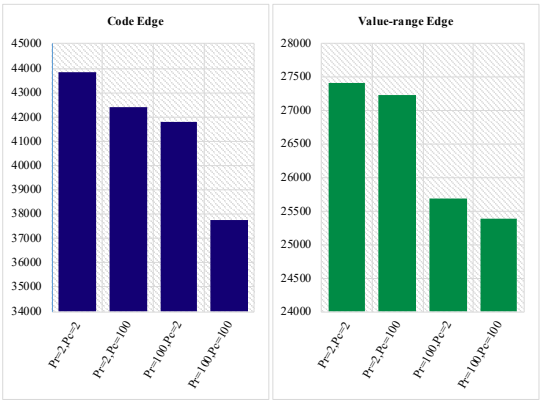


Figure 8: Either the number of code edges or the number of value-range edges discovered by `StateFuzz` declines significantly when P_r=100 or P_c=100.

```
1 syz_open_dev$dev_scull(dev ptr64[in, string["/dev/scull"]], id
      intptr, flags flags[open_flags]) fd_scull
2 ioctl$dev_scull_A(fd fd_scull, cmd const[0x41], arg ptr64[inout,
      string])
3 ioctl$dev_scull_B(fd fd_scull, cmd const[0x42], arg ptr64[inout,
      string])
4 ioctl$dev_scull_C(fd fd_scull, cmd const[0x43], arg ptr64[inout,
      string])
5 ioctl$dev_scull_V(fd fd_scull, cmd const[0x56], arg ptr64[inout,
      string])
```

Listing 3: System call templates for the example driver.

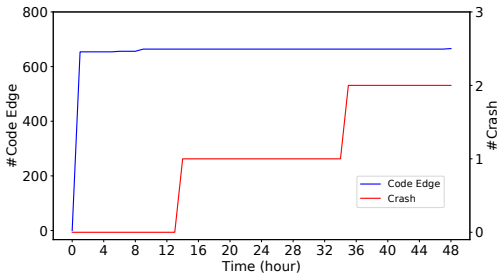


Figure 9: It takes Syzkaller 13 hours to find the first crash while fuzzing the motivation example driver. It is notable that it takes Syzkaller 4 hours to trigger the crash after covering all relevant code. Our solution `StateFuzz` triggers the out-of-bounds vulnerability with only 30,000 test cases being executed in less than 5 minutes.