FUZZING HARD-TO-COVER CODE

by

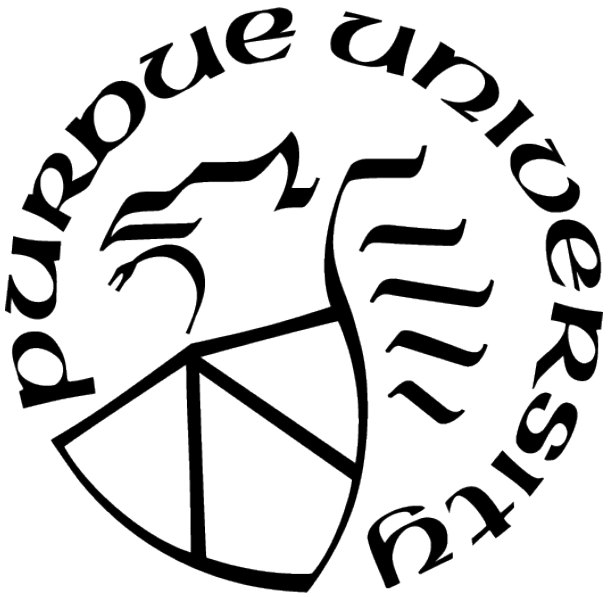
Hui Peng

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy



Department of Computer Science

West Lafayette, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Mathias Payer, Co-Chair

Department of Computer Science

Dr. Dave (Jing) Tian, Co-Chair

Department of Computer Science

Dr. Ninghui Li

Department of Computer Science

Dr. Dongyan Xu

Department of Computer Science

Dr. Pedro Fonseca

Department of Computer Science

Approved by:

Dr. Kihong Park

Head of the School Graduate Program

Dedicated to my parents and grandparents, and my sisters.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my major advisor Dr. Mathias Payer for giving me a precious chance to pursue my Ph.D. in the USA and for his invaluable guidance and support, as well as encouragement during my doctoral studies. I also would like to thank my co-advisor Dr. Dave (Jing) Tian for his invaluable help. To my committee members, Dr. Ninghui Li, Dr. Dongyan Xu and Dr. Pedro Fonseca, I would like to thank them for kindly agreeing to be on my committee, for their guidance on my work, and for their valuable time. I would like to thank past and present colleagues in the HexHive group, Nathan Burow, Scott Carr, Yuseok Jeon, Derrick McKee, Prashast Srivastava, Bader AlBassam, Sushant Dinesh, Priyam Biswas, Adrian Herrera, Kyriakos Ispoglou, Naif Almakhdhub, Abe Clements, Terry Hsu, Ahmed Hussein, Atri Bhattacharyya, Ahmad Hazimeh, Uros Tesic, Nicolas Badoux, Jelena Jankovic, Jean-Michel Crepel, Antony Vennard, Luca Di Bartolomeo, Ergys Dona, Lucio Romerio, Marcel Busch, Andr´es Sanchez and Zhiyuan Jiang for their precious feedback, insightful discussions, and friendship. I also would like to extend my gratitute to my collaborators outside Purdue, Dr. Yan Shoshitaishvili, Dr. Ardalan Amiri Sani, Zephyr Yao, for their invaluable help.

Last but not least, I would like to thank my family, for their love, trust, sacrifice, and support on this long journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

AFL	American Fuzzy Lop
ASAN	Address Sanitizer
KASAN	Kernel Address Sanitizer
MSAN	Memory Sanitizer
TSAN	Thread Sanitizer
OOM	Out Of Memory
UAF	Use After Free
OOB	Out Of Bound
USB	Universal Serial Bus
WebGL	Web Graphics Library
GPU	Graphics Processing Unit

ABSTRACT

Fuzzing is a simple yet effect approach to discover bugs by repeatedly testing the target system using randomly generated inputs. In this thesis, we identify several limitations in state-of-the-art fuzzing techniques: (1) the *coverage wall issue* , fuzzer-generated inputs cannot bypass complex sanity checks in the target programs and are unable to cover code paths protected by such checks; (2) *inability to adapt to interfaces* to inject fuzzer-generated inputs, one important example of such interface is the software/hardware interface between drivers and their devices; (3) *dependency on code coverage feedback*, this dependency makes it hard to apply fuzzing to targets where code coverage collection is challenging (due to proprietary components or special software design).

To address the coverage wall issue, we propose T-FUZZ, a novel approach to overcome the

issue from a different angle: by removing sanity checks in the target program. T-FUZZ leverages a coverage-guided fuzzer to generate inputs. Whenever the coverage wall is reached, a light-weight, dynamic tracing based technique detects the input checks that the fuzzer-generated inputs fail. These checks are then removed from the target program. Fuzzing then continues on the transformed program, allowing the code protected by the removed checks to be triggered and potential bugs discovered. Fuzzing transformed programs to find bugs poses two challenges: (1) removal of checks leads to over-approximation and false positives, and (2) even for true bugs, the crashing input on the transformed program may not trigger the bug in the original program. As an auxiliary post-processing step, T-FUZZ leverages a symbolic execution-based approach to filter out false positives and reproduce true bugs in the original program.

By transforming the program *as well as mutating the input*, T-FUZZ covers more code and finds more true bugs than any existing technique. We have evaluated T-FUZZ on the DARPA Cyber Grand Challenge dataset, LAVA-M dataset and 4 real-world programs (`pngfix`, `tiffinfo`, `magick` and `pdftohtml`). For the CGC dataset, T-FUZZ finds bugs in 166 binaries, Driller in 121, and AFL in 105. In addition, we found 4 new bugs in previously-fuzzed programs and libraries.

To address the inability to adapt to inferfaces, we propose USBFUZZ. We target the USB interface, fuzzing the software/hardware barrier. USBFUZZ uses device emulation to inject fuzzer-generated input to drivers under test, and applies coverage-guided fuzzing to device drivers if code coverage collection is supported from the kernel. In its core, USBFUZZ emulates an special USB device that provides data to the device driver (when it performs IO operations). This allows us to fuzz the input space of drivers from the device's perspective, an angle that is difficult to achieve with real hardware. USBFUZZ discovered 53 bugs in Linux (out of which 37 are new, and 36 are memory bugs of high security impact, potentially allowing arbitrary read or write in the kernel address space), one bug in FreeBSD, four bugs (resulting in Blue Screens of Death) in Windows and three bugs (two causing an unplanned restart, one freezing the system) in MacOS.

To break the dependency on code coverage feedback, we propose WEBGLFUZZER. To fuzz the WebGL interface (a set of JavaScript APIs in browsers allowing high-performance graphics rendering taking advantage of GPU acceleration on the device), where code coverage collection is challenging, we introduce WEBGLFUZZER, which internally uses a log guided fuzzing technique. WEBGLFUZZER is not dependent on code coverage feedback, but instead, makes use of the log messages emitted by browsers to guide its input mutation. Compared with coverage guided fuzzing, our log guided fuzzing technique is able to perform more meaningful mutation under the guidance of the log message. To this end, WEBGLFUZZER uses static analysis to identify which argument to mutate or which API call to insert to the current program to fix the internal WebGL program state given a log message emitted by the browser. WEBGLFUZZER is under evaluation and so far, it has found 6 bugs, one of which is able to freeze the X-Server.

1. INTRODUCTION

Fuzzing is an automated software testing technique that discovers faults by repeatedly testing the System Under Test (SUT) with randomly inputs. Fuzzing has been proven to be simple, yet effective [36, 69]. With the reduction of computational costs, fuzzing has become increasingly useful for both hackers and software developers, who use it to discover new bugs/vulnerabilities in software. As such, fuzzing has become a standard in software development to improve reliability and security [46, 71].

According to how inputs are produced, fuzzers can be roughly divided into two categories: generational fuzzers and mutational fuzzers. Generational fuzzers, such as PROTOS [99], SPIKE [2], and PEACH [29], construct inputs according to some provided format specification. By contrast, mutational fuzzers, including AFL [135], honggfuzz [44], and zzuf [64], create inputs by randomly mutating seeds which are either analyst provided or randomly generated. Generational fuzzing requires an input format specification, which imposes significant manual effort to create (especially when attempting to fuzz software on a large scale) or may be infeasible if the format is not available. Thus, most recent work in the field of fuzzing, including our work, focuses on mutational fuzzing. In this work, when using fuzzing we refer to mutational fuzzing.

The state-of-art fuzzing technique is coverage-guided fuzzing. Essentially, coverage-guided fuzzing uses evolutionary algorithm [104] based on code coverage feedback from the target system to sample the input space. Specifically, it keeps track of a set of interesting inputs that triggered new code coverage, and focuses on mutating the interesting inputs when generating new inputs. It has been shown to be very effective [109, 136] in bug finding as demonstrated by coverage-guided fuzzers like AFL (American Fuzzy Lop) [135] and honggfuzz [44].

We focus on coverage-guided mutational greybox fuzzing as this has become the de-facto fuzzing standard due to its ability to quickly adapt to varying situations without the need for heavy-weight configuration.

1.1 Problems with existing fuzzing techniques

Fuzzing, though simple and capable of finding many vulnerabilities in real-world programs, faces challenges when applied in increasing complex software systems. In this thesis, we identify the following limitations of state-of-art fuzzing tools based on our extensive studies and experiments on real-world programs. These limitations serve as motivation of the work in this thesis.

1. The *coverage wall* issue. The coverage wall is a hard limit in the coverage that a fuzzer can achieve in a target program. Overcoming the *coverage wall* is a fundamental challenge in all mutational fuzzers. Regardless of the mutation strategy, whether it be a purely randomized mutation or coverage-guided mutation, it is highly unlikely for the fuzzer to generate inputs that can bypass complex sanity checks in the target program. This is because, due to their simplicity, mutational fuzzers are ignorant of the actual input format expected by the program. This inherent limitation prevents mutational fuzzers from triggering code paths protected by sanity checks and finding "deep" bugs hidden in such code.

2. *Fuzzing hard-to-adapt interfaces.* A fuzzer works by repeatedly testing the target program using fuzzer-generated random data as input. Input injection is a problem ignored by existing fuzzers because they target exclusively programs taking inputs through file system or other software interfaces, which can be easily adapted to take fuzzer-generated-inputs (e.g., by modifying the content of a file passed to the target program). However, input injection through some interfaces is challenging. An example is the software/hardware interface between the drivers and devices. As the drivers are implemented to take input from hardware devices directly, and the hardware devices is not amenable to external control, fuzzing the input space from the device side is challenging.

3. *Dependency on code coverage.* The input space of a target program is typically huge, thus dumb fuzzing can only explore shallow code paths. To effectively reduce the search space, state-of-art coverage guided fuzzing techniques use an evolutionary algorithm

based on code coverage as feedback to guide their input mutation [32, 47, 101, 135]. Dependency on code coverage makes it difficult to be applied on targets in which code coverage collection is challenging due to proprietary software component, or special software designs (e.g., the target code is designed to run in multiple processes).

1.2 Thesis Statement

This thesis addresses limitations of state-of-art fuzzing techniques and aid the analysis of code that is hard to be covered. The thesis statement is:

> *Current fuzzing strategies can be improved to reach hard-to-cover and security critical code by such techniques as program transformation, symbolic execution, device emulation, and static analysis.*

This statement is demonstrated through three pieces of work. (1) To overcome coverage wall, T-FUZZ uses program transformation to open code guarded complex sanity checks in the target program to fuzzer generated inputs. Crashes detected on transformed programs are then verified using a symbolic execution based technique. (2) To fuzz the device input space of USB device drivers, USBFUZZ leverages a software implemented USB device in a virtual machine to inject the fuzzer generated inputs to the drivers under test. (3) To fuzz the WebGL interface, where coverage collection is challenging, WEBGLFUZZER uses a novel log guided fuzzing technique, which is highly customized to the WebGL (Web Graphics Library, a JavaScript interface in browsers for high performance graphics rendering leveraging the underlying GPU support) implementation of Chrome browser, eliminates dependency on code coverage and guided by runtime log messages to perform more meaningful mutations than coverage guided fuzzing.

Each of listed work is summarized below and full details are provided in subsequent chapters.

1.2.1 T-FUZZ: Overcoming Coverage Wall by Program Transformation

To overcome the coverage wall, most existing approaches rely on imprecise heuristics or complex and expensive program analysis (e.g., symbolic execution or taint analysis) techniques to generate and/or mutate inputs to bypass the sanity checks. For example, AFL [135], the most widely used coverage guide fuzzer, uses coverage feedback to heuristically infer the values and positions of the magic values in the input. Several recent approaches [66, 96, 106, 128] leverage symbolic analysis or taint analysis to improve coverage by generating inputs to bypass the sanity checks in the target program. However, limitations persist — as we will show, state-of-the-art techniques such as AFL [135] and Driller [106] find vulnerabilities in less than half of the programs in a popular vulnerability analysis benchmarking dataset (the challenge programs from the DARPA Cyber Grand Challenge).

Recent research into fuzzing techniques focuses on finding new ways to generate and evaluate inputs. However, there is no need to limit mutation to program inputs alone. In fact, the *program itself* can be mutated to assist bug finding in the fuzzing process. Following this intuition, we propose *Transformational Fuzzing*, a novel fuzzing technique aimed at improving the bug finding ability of a fuzzer by disabling input checks in the program. This technique turns the dynamic code coverage problem on its head: rather than necessitating time-consuming and heavyweight program analysis techniques to generate test cases to bypass complex input checks in the code, we simply detect and disable these sanity checks. Fuzzing the transformed programs allows an exploration of code paths that were previously-protected by these checks, discovering potential bugs in them.

Of course, removing certain sanity checks may break the logic of the original program and the bugs which are found in transformed programs may thus contain *false positives*, potentially overwhelming the analyst. To remove false positives, we develop a post-processing symbolic execution-based analysis. The remaining inputs reproduce true bugs in the original program. Though this method is complex and heavyweight (like test case mutation techniques of related work), it only needs to be done to verify detections after the fact, and (unlike existing test case mutation techniques) does not slow down the actual analysis itself.

To show the usefulness of transformational fuzzing, we developed a prototype named T-FUZZ. At its base, it employs an off-the-shelf coverage-guided fuzzer to explore a program. Whenever the fuzzer can no longer generate new inputs to trigger unexplored code paths, a lightweight dynamic tracing-based approach discovers all input checks that the fuzzer-generated inputs failed to satisfy, and the program is transformed by selectively disabling these checks. Fuzzing then continues on the transformed programs.

In comparison to existing symbolic analysis based approaches, T-FUZZ excels in two aspects: (1) better scalability: by leveraging lightweight dynamic tracing-based techniques during the fuzzing process, and limiting the application of heavyweight symbolic analysis to detected crashes, the scalability of T-FUZZ is not influenced by the need to bypass complex input checks; and (2) the ability to cover code paths protected by "hard" checks.

To determine the relative effectiveness against state-of-the-art approaches, we evaluated T-FUZZ on a dataset of vulnerable programs from the DARPA Cyber Grand Challenge (CGC), the LAVA-M dataset, and four real-world programs relying on popular libraries (`pngfix/libpng`, `tiffinfo/libtiff`, `magick/ImageMagick` and `pdftohtml/libpoppler`). In the CGC dataset, T-FUZZ finds bugs in 166 binaries out of 296, improving over Driller [106] by 45 binaries and over AFL by 61 binaries, and demonstrating the effectiveness of transformational fuzzing. Evaluation of the LAVA-M dataset shows that T-FUZZ significantly outperforms Steelix and VUzzer in the presence of "hard" input checks, such as checksums. The ground truth provided by these two datasets allows us to determine that our tool is able to filter out false positives at the cost of surprisingly few false negatives (6%-30%). Finally, the evaluation of T-FUZZ on real-world applications leads to the discovery of 3 new bugs.

1.2.2 USBFUZZ: Fuzzing Software/Hardware Interface by Device Emulation

The challenge in fuzzing the software/hardware interface lies in providing random input, because the driver code is implemented to interact directly with the hardware. The device cannot be easily modified to provide unexpected input. Dedicated programmable devices for certain device types (specially designed for testing drivers, e.g., FaceDancer [41] is the only available types of such device for USB interface to the best of knowledge, no such

devices available for other interfaces) can be used to fuzz the exposed hardware interface but they are expensive and unscalable (as one device can only be used in one fuzzing instance). More importantly, it is challenging to automate fuzzing on real hardware due to the required physical actions (attaching and detaching the device) for each test. Some solutions adapt the kernel. E.g., the kernel fuzzer syzkaller [47] contains a usb-fuzzer [42] extension which implements a dummy USB device using the gadgetfs [81] framework and connects it to a USB device driver through the software implemented controller (dummy hcd [107]). This approach is not portable (as it is tightly coupled with the kernel implementation), requires a deep understanding of the hardware specification and kernel implementation. In addition, due to limitations in dummy hcd, some features (e.g., isochronous endpoints [34]) in the USB protocol are not supported, and thus drivers using these features cannot be tested.

We propose a device-emulation based approach to fuzz the peripheral input space of the kernel. At its core, USBFUZZ uses an emulated fuzzing device to provide random input to a virtualized kernel. The target kernel is run in a hypervisor. In each iteration, a coverage-guided fuzzer runs a test by virtually attaching the emulated fuzzing device to the target system, which forwards the fuzzer generated inputs to the target kernel. After the test finishes, kernel code coverage is collected to guide the input mutation. Compared with existing solutions, the device emulation based approach is cheap (no incuring any hardware cost), portable (working in the device layer thus straightforward to be ported to different platforms) and scalable (multiple instances of the device can be started by running emulators).

Given the versatility of the USB bus, a plethora of peripheral devices (e.g., HID, CDC, Audio, or Video) rely on it. Almost all computer systems support USB devices. Compared to other peripherals, systems that expose buggy drivers on the USB bus are easily attacked by rewriting the firmware of a benign device or through a programmable USB device like FaceDancer [41]. Thus we implemented a prototype called USBFUZZ that focuses on USB drivers, though our design of emulating compromised hardware through a fuzzer device is generic.

To verify its effectiveness, we evaluated USBFUZZ on the USB device drivers in Linux kernel, FreeBSD, Windows and MacOS. On Linux, we apply coverage guided fuzzing based

on static instrumentation. We use a communicating device in the guest system to expose a memory area in the the fuzzer process to the (target) guest system and adapted KCOV [125] to collect code coverage of target driver code. On FreeBSD, MacOS, and Windows, we apply dumb fuzzing through cross-pollination seeded by the Linux inputs. USBFuzz discovered a total of 26 new bugs, including 16 memory bugs of high security impact in various Linux subsystems (USB core, USB sound, and network), one bug in FreeBSD, three in MacOS (two resulting in an unplanned reboot and one freezing the system), and four in Windows 8 and Windows 10 (resulting in Blue Screens of Death), and one bug in the Linux USB host controller driver and another one in a USB camera driver. From the Linux bugs, we have fixed and upstreamed 11 bugs and received 10 CVEs.

1.2.3 WEBGLFUZZER: Fuzzing WebGL Interface through Log Guided Mutation

WebGL is a new set of standardized Javascript APIs allowing GPU accelerated graphics rendering for the web platform. It is now widely supported in many web platforms, including traditional Web tools (browsers for destktop and mobile OSes) and much newer application frameworks (e.g., Android WebView [22], iOS WKWebView [26], Electron [30]). The introduction of WebGL brings a lot of security concerns as it exposes the underlying native graphics stack to remote and untrusted users. Many recent CVEs have been reported in this interface.

Although widely supported and security critical, little work has been done to address its security risks. In an attempt to apply state-of-art coverage guide fuzzing to this interface, we face the the following challenges: (1) difficulty in collecting code coverage, because the software stack consists of several layers of libraries, different processes, and code running both in user and kernel space, some of the components are closed source and proprietary; (2) code coverage does not indicate how to perform mutation on the current input to effectively extend coverage.

We propose a novel, log message guided fuzzing technique whose input mutation is guided by runtime log messages emitted by browsers, instead of code coverage. The idea of log guided fuzzing technique is based on a key observation that browsers emit meaningful log

messages to aid developers in debugging WebGL programs, when errors are detected in the WebGL program. We leverage these log messages to guide the input mutation. To this end, we analyze the dependencies of message logging statement to identify target argument or dependent APIs, and focus mutation on the identified argument or using dependent APIs, instead of performing random mutations. Further, to leverage semantic meaning of log messages, we use build mutating rules based on natural language analysis on the log messages.

We have implemented log message guided fuzzing technique in a prototype called WEBGLFUZZER and are performing evaluation in popular browsers (Chrome, Firefox and Safari) on both desktop (Linux, Windows and MacOS) and mobile (Andrdoid and iOS) OSes, so far, we have found 6, 3 in Chrome, 2 in Safari, and 1 in Firefox, one of the bugs in Chrome freezes the X-Server in Linux.

1.3 Contributions

The goal of the thesis is to aid the analysis of code that is hard to be covered by state-of-art fuzzing tools. The work presented in this dissertation has been peer-reviewed and published, except for WEBGLFUZZER which is still in preparation for submission. In particular, our program transformation based fuzzing tool, T-FUZZ was publised at IEEE Symposium on Security and Privacy 2018; our device emulation based fuzzing technique, USBFUZZ was published at Usenix Security Symposium 2020.

In summary, the core contributions of this thesis are as follows:

- A novel program transformation based fuzzing technique that finds more 45% - 58% more bugs in the CGC dataset and 4 bugs in real-world programs than the best related work at the time of publication.

- A novel device emulation based driver fuzzing technique. Our prototype targeting USB drivers has led to the discovery of more than 50 bugs in the linux kernel, one bug in FreeBSD, four bugs (resulting in Blue Screens of Death) in Windows and three bugs (two causing an unplanned restart, one freezing the system) in MacOS.

- A novel log guided fuzzing technique customized for WebGL interface in browsers. Our log guided fuzzing technique is not dependent on code coverage collection which is challenging in WebGL implementation, and capable of making meaningful mutations on the inputs.

2. T-FUZZ: FUZZING BY PROGRAM TRANSFORMATION

2.1 Coverage Wall & T-FUZZ Motivation

Coverage wall is a certain limit in the coverage that a fuzzer can achieve in a target program because the inputs inherently randomly generated by the fuzzer are highly unlikely to bypass all the complex sanity checks in the target program. When fuzzers fail to generate inputs to bypass the sanity checks, they become "stuck" and continue to generate random inputs without covering new code paths.

As an example, Figure 2.1 shows a secure compressed file format: the first 4 bytes are a file header which contains hardcoded magic values ("SECO"); the *Keys* field declares 95 unique chars whose ASCII values must be within the range of [32, 126]; the *Len* field specifies the length of the following compressed data; and finally a 4-byte *CRC* field to integrity check the compressed data. The example is based on CGC KPRCA_00064, extended with a CRC check.

Listing 2.1 is a program that parses and decompresses the compressed file format shown above. It has a "deep" stack buffer overflow bug in *decompress* function in line 31. Before calling *decompress*, the program performs a series of checks on the input:

1. check on the magic values of the header field in line 8.
2. check for range and uniqueness on the next 95-byte Keys field in line 13-18.
3. check on the CRC field for potential data corruption in line 24.

If any of these checks fail, the input is rejected without calling *decompress*, thereby not triggering the bug.

These checks highlight the challenges in mutational fuzzers and related techniques. First of all, it takes a lot of effort for a mutational fuzzer like AFL [135] to bypass C1 without

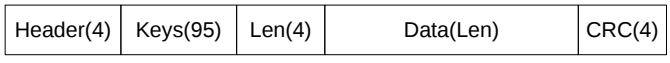


Figure 2.1. Secure compressed file format

```c
#define KEY_SIZE  95
int sc_decompress(int infd, int outfd) {
  unsigned char keys[KEY_SIZE];
  unsigned char data[KEY_SIZE];
  char *header = read_header(infd)
  // C1: check for hardcoded values
  if (strcmp(header, "SECO") != 0)
    return ERROR;
  read(infd, keys, KEY_SIZE);
  memset(data, 0, sizeof(data));
  // C2: range check and duplicate check for keys
  for (int i = 0; i < sizeof(data); ++i) {
    if (keys[i] < 32 || keys[i] > 126)
      return ERROR;
    if (data[keys[i] - 32]++ > 0)
      return ERROR;
  }
  unsigned int in_len = read_len(infd);
  char *in = (char *) malloc(in_len);
  read(infd, in, in_len);
  unsigned int crc = read_checksum(infd);
  // C3: check the crc of the input
  if (crc != compute_crc(in, in_len)) {
    free(in);
    return ERROR;
  }
  char *out;
  unsigned int out_len;
  // Bug: function with a stack buffer overflow bug
  decompress(in, in_len, keys, &out, &out_len);
  write(outfd, out, out_len);
  return SUCCESS;
}
```

Listing 2.1. An example containing various sanity checks

the help of other techniques. As mutational fuzzers are unaware of the file format, they will struggle to bypass C2 or C3. Additionally, although symbolic analysis based approaches like Driller [106] can quickly bypass C1 and C2 in this program, they will fail to generate accurate inputs to bypass C3 as the constraints derived from the checksum algorithm are too complex for modern constraint solvers [14].It is therefore unlikely that the buggy *decompress* function will be triggered through either mutational fuzzing or symbolic analysis.

Regarding sanity checks in the context of fuzzing, we make the following observations:

1. Sanity checks can be divided into two categories: **NCC** (Non-Critical Check) and **CC** (Critical Check). NCCs are those sanity checks which are present in the program logic to filter some orthogonal data, e.g., the check for a magic value in the decompressor example above. CCs are those which are essential to the functionality of the program, e.g., a length check in a TCP packet parser.

2. NCCs can be removed without triggering spurious bugs as they are not intended to prevent bugs. Removal of NCCs simplifies fuzzing as the code protected by these checks becomes exposed to fuzzer generated inputs. Assume we remove the three checks in the decompressor above, producing a transformed decompressor. All inputs generated by the fuzzer will be accepted by the transformed decompressor and the buggy *decompress* function will be covered and the bug found.

3. Bugs found in the transformed program can be reproduced in the original program. In the decompressor above, as the checks are not intended for preventing the stack buffer overflow bug in the decompress function, bugs found in the transformed decompressor are also present in the original decompressor. Assume that the fuzzer found a bug in the transformed decompressor with a crashing input X, it can be reproduced in the original decompressor by replacing the Header, Keys, and CRC fields with values that satisfy the check conditions in the program.

4. Removing CCs may introduce spurious bugs in the transformed program which may not be reproducible in the original program. These false positive bugs need to be

filtered out during a post-processing phase to ensure that only the bugs present in the original program are reported.

NCCs are omnipresent in real-world programs. For example on Unix systems, all common file formats use the first few bytes as magic values to identify the file type. In network programs, checksums are widely used to detect data corruption.

Based on these observations, we designed T-FUZZ to improve fuzzing by detecting and removing NCCs in programs. By removing NCCs in the program, the code paths protected by them will be exposed to the fuzzer generated inputs and potential bugs can be found. T-FUZZ additionally helps fuzzers cover code protected by "hard" sanity checks like C3 in Listing 2.1.

2.2 T-FUZZ Design

Figure 2.2 depicts the main components and overall workflow of T-FUZZ. Here we summarize its main components, the details will be covered in the following section.

Fuzzer: T-FUZZ uses an existing coverage-guided fuzzer, e.g., AFL [135] or honggfuzz [44], to generate inputs. T-FUZZ depends on the fuzzer to keep track of the paths taken by all the generated inputs and realtime status information regarding whether it is "stuck". As output, the fuzzer produces all the generated inputs. Any identified crashing inputs are recorded for further analysis.

Program Transformer: When the fuzzer gets "stuck", T-FUZZ invokes its Program Transformer to generate transformed programs. Using the inputs generated by the fuzzer, the Program Transformer first traces the program under test to detect the NCC candidates and then transforms copies of the program by removing certain detected NCC candidates.

Crash Analyzer: For crashing inputs found against the transformed programs, the Crash Analyzer filters false positives using a symbolic-execution based analysis technique.

Algorithm 1 shows how a program is fuzzed using T-FUZZ. First, T-FUZZ iteratively detects and disables NCC candidates that the fuzzer encounters in the target program. A

queue (`programs`) is used to save all the programs to fuzz in each iteration, and initially contains the original program.

In each iteration, T-FUZZ chooses a program from the queue and launches a fuzzer process (in the algorithm, the invocation of `Fuzzer`) to fuzz it until it is unable to generate inputs that further improve coverage. Using inputs generated by the fuzzer before it gets "stuck", T-FUZZ detects additional NCC candidates and generates multiple transformed programs (by invoking `Program_Transformer`) with different NCCs disabled. The transformed programs are added to the queue for fuzzing in the following iterations. All crashes found by the Fuzzer in each iteration are post-processed by the Crash Analyzer to identify false positives.

Algorithm 1: Fuzzing with T-FUZZ

Input: *program*: original program

1 $programs \leftarrow \{program\}$
2 **while** $programs \neq \emptyset$ **do**
3 $\quad p \leftarrow Choose\_Program(programs)$
4 $\quad inputs \leftarrow Fuzzer(p)$
5 $\quad programs \leftarrow programs \cup Program\_Tranformer(p, inputs)$

2.2.1 Detecting NCC Candidates

To detect the NCCs in a program, different options are available with varying precision and overhead. For example, we can use complex data flow and control flow analysis to track dependencies between the sanity checks in the program and input. This approach has good precision, but involves very high overhead (which is extremely detrimental for fuzzing, as fuzzers are heavily optimized for performance), and often needs to be based on fairly brittle techniques (which is detrimental to the applicability of the technique). Considering this, in T-FUZZ, we use a less precise, but lightweight approach that approximates NCCs; we use the set of checks that could not be satisfied by *any* fuzzer-generated inputs when the fuzzer gets stuck.

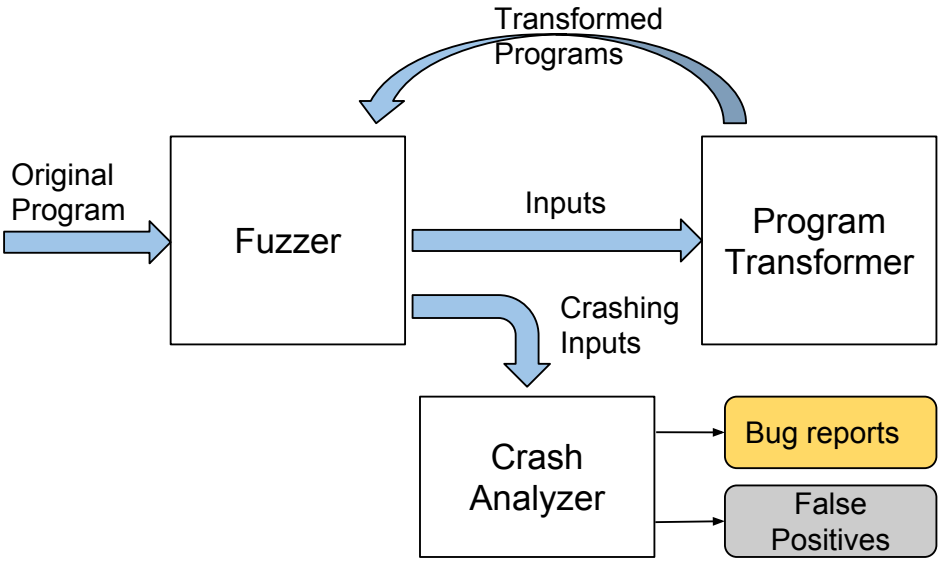


Figure 2.2. Overview of T-FUZZ

Sanity checks are compiled into conditional jump instructions in the program, and represented as a source basic block S with 2 outgoing edges in the control flow graph (CFG)[1]. Each outgoing edge corresponds to either True or False of the condition, which are denoted as T and F respectively. Failing to bypass a sanity check means that only the T or F edge is ever taken by any fuzzer-generated input.

In T-FUZZ we use all the boundary edges in the CFG — the edges connecting the nodes that were covered by the fuzzer-generated inputs and those that were not — as *approximation* of NCC candidates. Denote all the nodes in the CFG of the program as N and all the edges as E, and let CN be the union of executed nodes, CE be the union of taken edges by all inputs in I. The boundary edges mentioned above can be formalized as all edges e satisfying the following conditions:

1. e is not in CE;
2. source node of e is in CN;

Algorithmically, T-FUZZ first collects the cumulative node and edge coverage executed by the fuzzer generated inputs. Then, it uses this cumulative coverage information to calculate the NCC candidates. T-FUZZ uses a dynamic tracing based approach to get the cumulative edge and node coverage for a set of inputs. As shown in Algorithm 3, for each input i, it invokes *dynamic_trace* to get the sequence of executed edges under input i. The union of edges and nodes in all traces of inputs is returned.

Algorithm 2 shows how NCC candidates are detected based on the cumulative edge and node coverage collected by Algorithm 3. It builds the CFG of the program and then iterates over all the edges, returning those that satisfy the conditions shown above as NCC candidates.

We use the following example to demonstrate the effect of this algorithm. Listing 2.2 takes 16 bytes as input and uses the first two bytes as magic values and the third byte to decide whether to use `format1` or `format2` to process the input. Figure 2.3 shows the CFG of the program.

[1]↑To simplify the discussion we assume that switch statements are compiled to a tree of if conditions, instead of a jump table, although the tool itself makes no such assumption.

Algorithm 2: Detecting NCC candidates

Input: $program$: The binary program to be analyzed
Input: CE: cumulative edge coverage
Input: CN: cumulative node coverage

```
1 cfg ← CFG(program)
2 NCC ← ∅
3 for e ∈ cfg.edges do
4     if e ∉ CE ∧ e.source ∈ CN then
5         NCC ← NCC ∪ {e}
```

Output: NCC: detected NCC candidates

Algorithm 3: Calculating cumulative edge and node coverage

Input: $inputs$: inputs to collect cumulative coverage

```
1 CE ← ∅
2 CN ← ∅
3 for i ∈ inputs do
4     trace ← dynamic_trace(i)
5     for e ∈ trace do
6         CE ← CE ∪ {e}
7         CN ← CN ∪ {(e.source, e.destination)}
```

Output: CE: cumulative edge coverage
Output: CN: cumulative node coverage

Assume the fuzzer component has generated a set of inputs {"123...", "A12..", "AB_...", "AB{..."}, and gets "stuck" without being able to cover `format1` and `format2` we are interested in. Running our algorithm we can easily detect the sanity checks that are guarding the invocation of `format1` and `format2`. As "123..." triggers execution path A→H, "A12.." triggers execution path A→B→H, "AB_.." triggers A→B→C→E→G→H, and "AB{.." triggers A→B→C→D→H, the cumulative node and edge coverage are {A, B, C, D, E, G, H} and { A→H, A→B, B→H, B→C, C→D, D→H, C→E, E→G, G→H} (see Figure 2.4), and the detected NCC candidates are {C→D, E→H, G→I} (see Figure 2.5). Obviously D→F and G→I are the sanity checks that preventing the fuzzer generated inputs to cover `format1` and `format2`.

```
1  void main() {
2    char x[16];
3    read(stdin, x, 16);
4
5    if (x[0] == 'A' && x[1] == 'B') {
6        if (x[2] >= 'a') {
7          if (x[2] <= 'z') {
8             format1(x);
9          } else {
10           goto failure;
11         }
12       } else {
13         if (x[2] >= 'A' && x[2] <= 'Z') {
14            format2(x);
15         } else {
16            goto failure;
17         }
18       }
19   }
20 failure:
21   error();
22 }
```

Listing 2.2. An example demonstrating the effect of NCC detection algorithm

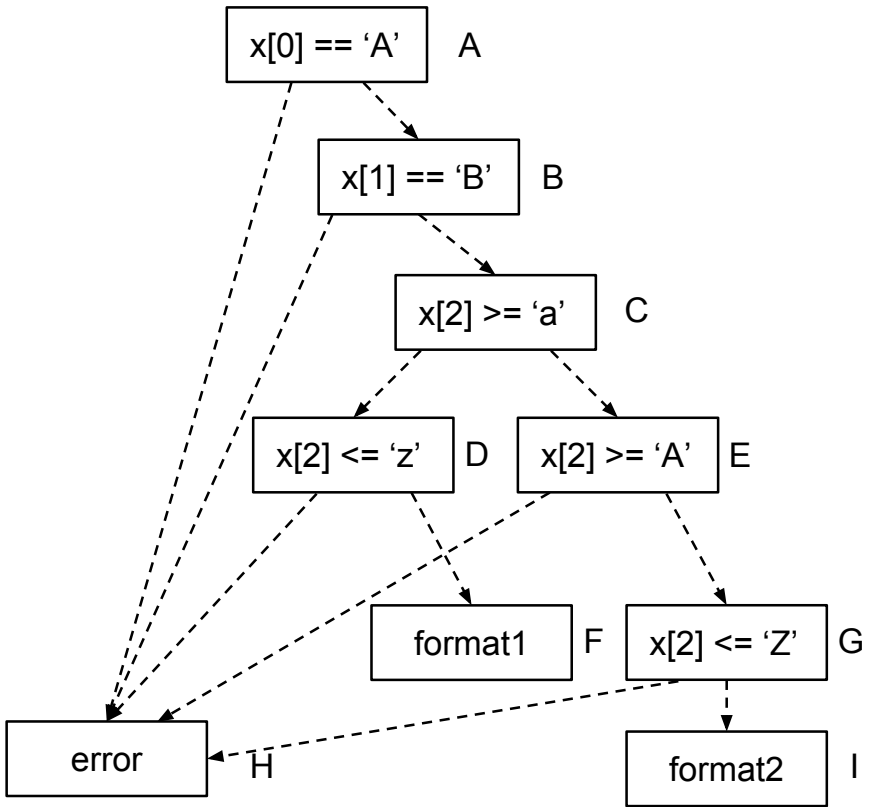


Figure 2.3. CFG of example program in Listing 2.2.

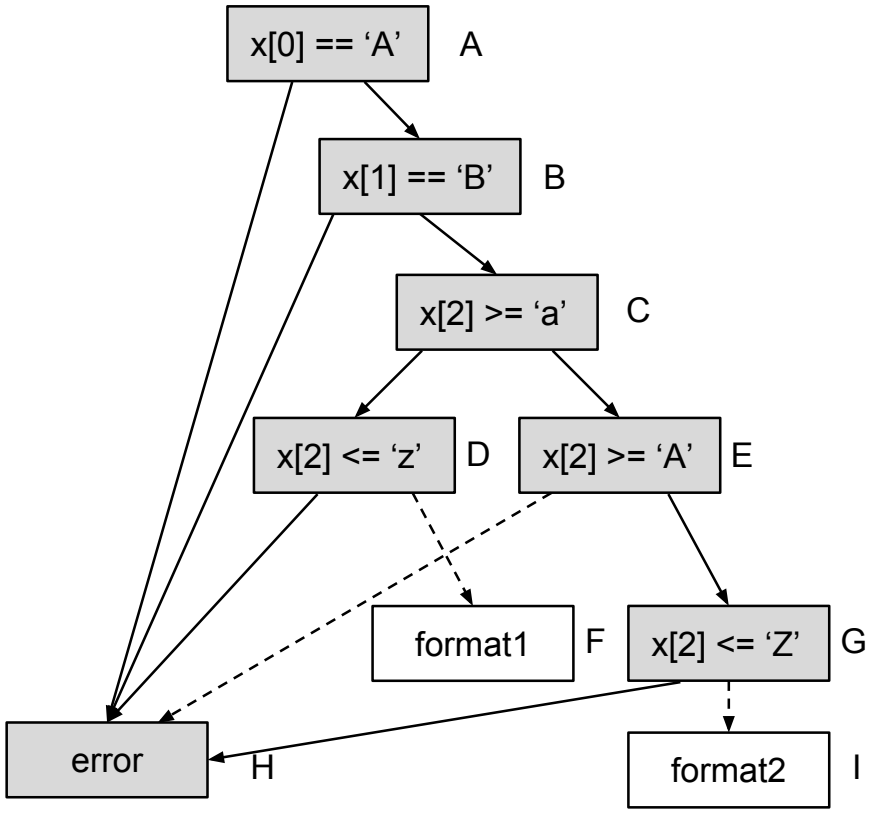


Figure 2.4. Cumulative edge and node coverage of input set { “123...”, “A12...”, “AB]..”, “AB{...”}. Cumulative nodes are greyboxes and cumulative edges are solid arrows.

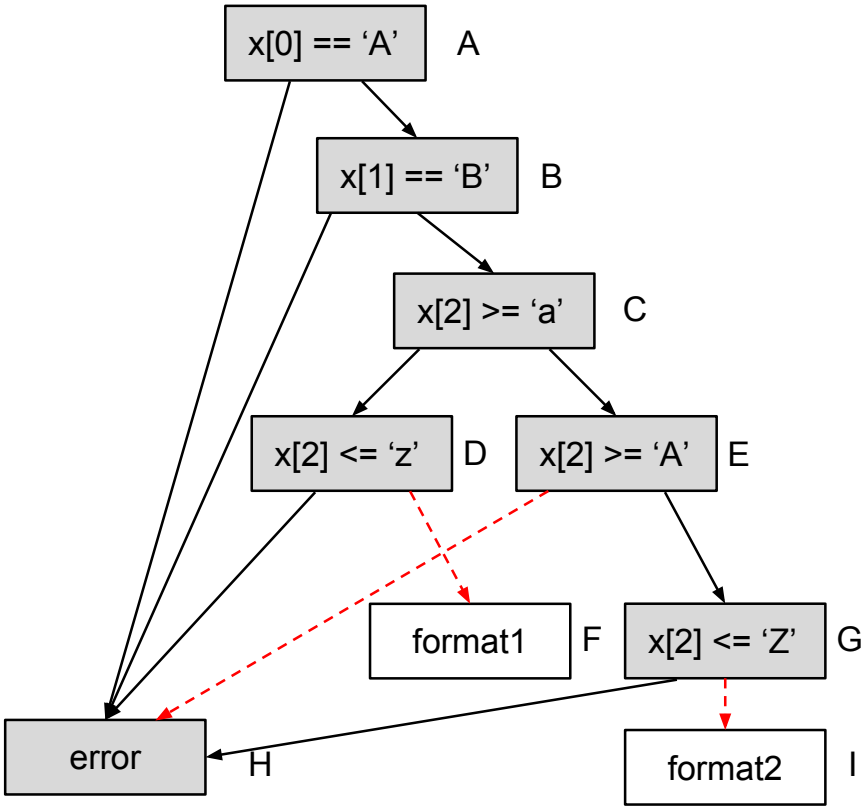


Figure 2.5. NCC candidates detected (red dashed arrows).

```
1  void main() {
2    char x[10];
3
4    if (read(0, x, 10) == -1)
5      goto failure;
6    // main logic for processing x
7    ...
8    return;
9  failure:
10   error();
11 }
```

Listing 2.3. An program showing check for error code

2.2.2 Pruning Undesired NCC Candidates

The NCC candidates detected using the algorithm in Section 2.2.1 is an over-approximation of sanity checks and may contain undesired checks. Before feeding the NCC candidates from Algorithm 2 into the Program Transformer, a filtering step prunes any undesired candidates that we deem unlikely to help bug finding. We list the types of undesired checks we encountered, the consequences of removing these checks, and our approaches to remove undesired checks.

Algorithm 2 in Section 2.2.1 detects NCC candidates in all executed (or not executed) code. When fuzzing, we are often interested in just the program executable or a specific library. In a first step, we therefore prune any candidates that are not in the desired object.

The second source of undesired checks are checks for error codes that immediately terminate the program. For example in the program shown in Listing 2.3, the return value of the *read* system call is checked for possible errors in line 4. As read errors happen infrequently, the error checking code is not executed and thus detected as NCC candidate.

Treating these checks as NCCs does not result in useful detections from T-FUZZ. Consider the detected check shown in Listing 2.3. Removing the check results in a program execution where only the error handling code is executed before the program is abnormally terminated. It is unlikely for the fuzzer to find bugs along this path.

Given that the program often terminates with very short code paths after detecting such a severe error, we heuristically use the number of basic blocks following the detected NCC

candidate as the length of code paths and define a threshold value to tell an error handling code path.

The intuition behind this approach is to focus on NCCs that result in a large amount of increased coverage compared to NCCs that immediately terminate the program under test (due to, e.g., a severe error).

2.2.3 Program Transformation

We considered different options to remove detected NCC candidates, including as dynamic binary instrumentation, static binary rewriting, and simply flipping the condition of the conditional jump instruction. Dynamic binary instrumentation often results in high overhead and static binary translation results in additional complexity due to the changed CFG. On the other hand, flipping conditions for conditional jumps is straight-forward and neutral to the length of the binary, providing the advantages of static rewriting without the need for complex program analysis techniques. This technique maintains the inverted path condition in the program, and the path condition in the original program can be easily recovered.

T-FUZZ transforms programs by replacing the detected NCC candidates with a negated conditional jump. Doing so maintains the structure of the original program while keeping necessary information to recover path conditions in the original program. As the addresses of the basic blocks stay the same in the transformed program, the traces of the transformed program directly map to the original program. Maintaining the trace mapping greatly reduces the complexity of analyzing the difference between the original program and transformed program in the Crash Analyzer.

Algorithm 4 shows the pseudo code of the Program Transformer. The Program Transformer takes a program to transform and NCC candidates to remove as input. As there is at most one jump instruction in a basic block, it simply scans all the instructions in the source block of the NCC candidate and overwrites the first conditional jump instruction with its negated counterpart instruction. To keep track of the modified conditional jump (by invocation of *negate_conditional_jump*), the addresses of modified instructions are passed in as

argument, and the address of each modified instruction is recorded and returned as part of the output.

Algorithm 4: Transforming program

Input: *program*: the binary program to transform
Input: *c_addrs*: the addresses of conditional jumps negated in the input program
Input: *NCC*: NCC candidates to remove

```
1 transformed_program ← Copy(program)
2 for e ∈ NCC do
3     basic_block ← BasicBlock(transformed_program, e.source)
4     for i ∈ basic_block do
5         if i is a conditional jump instruction and i.addr ∉ c_addrs then
6             negate_conditional_jump(program, i.addr)
7             c_addrs ← c_addrs ∪ {i.addr}
8             break
```

Output: *transformed_program*: the generated program with NCC candidate disabled
Output: *c_addrs*: the locations modified in the transformed program

2.2.4 Filtering out False Positives and Reproducing Bugs

As the removed NCC candidates might be meaningful guards in the original program (as opposed to, e.g., magic number checks), removing detected NCC edges might introduce new bugs in the transformed program. Consequently, T-FUZZ's Crash Analyzer verifies that each bug in the transformed program is also present in the original program, thus filtering out false positives. For the remaining true positives, an example input that reproduces the bug in the original program is generated.

The Crash Analyzer uses a *transformation-aware* combination of the preconstrained tracing technique leveraged by Driller [106] and the Path Kneading techniques proposed by ShellSwap [7] to collect path constraints of the original program by tracing the program path leading to a crash in the transformed program. The satisfiability of the collected path constraints indicates whether the crash is a false positive or not. If the path constraints are satisfiable, the Crash Analyzer reproduces the bug in the original program by solving the path constraints.

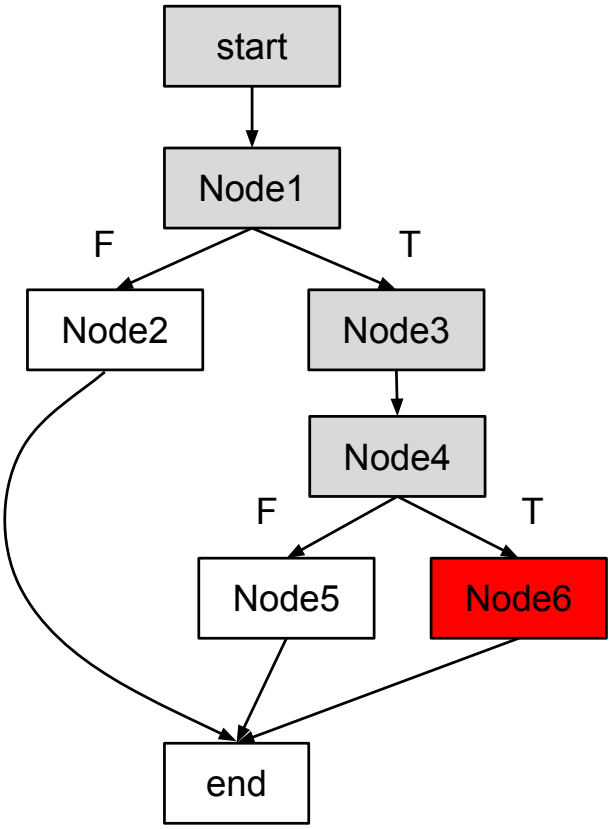


Figure 2.6. An example of transformed program

To illustrate the idea behind the algorithm, we show a transformed program P whose CFG is represented in Figure 2.6, and a crashing input I. I executes a code path shown as grey nodes in Figure 2.6, with the crash in Node6 at address CA. Node1 and Node4 contain conditional jumps that are negated by the Program Transformer and because of this, the T edges are taken when executing I. The constraints associated with NCCs in Node1 and Node4 are denoted as C_1 and C_4 respectively.

When the Crash Analyzer traces the transformed program, it maintains two sets of constraints: one for keeping track of the constraints in the transformed program (denoted as CT) the other for keeping track of that in the original program (denoted as CO). Before the tracing starts, I is converted to a preconstraint (denoted as PC) and added to CT, this ensures that the trace will follow the code path shown in Figure 2.6. While tracing the transformed program, if the basic block contains a negated conditional jump, the inverted

path constraint associated with the conditional jump is added to CO, otherwise, the path constraints are added to CO. In this example, $\neg C_1$ and $\neg C_4$ are added to CO. When the tracing reaches the crashing instruction in the program, the cause of the crash (denoted as CC) is encoded to a constraint and also added to CO. For example, if it is an out-of-bound read or write, the operand of the load instruction is used to encode the constraint, if it is a divide by zero crash, the denominator of the div instruction is used to encode the constraint. If the path constraints in CO can be satisfied, it means that it is possible to generate an input that will execute the same program path and trigger the same crash in the original program. Otherwise it is marked as a false positive.

The pseudo code of the Crash Analyzer is shown in Algorithm 5. It takes the transformed program and the addresses of negated conditional jumps, a crashing input and the crash address in the transformed program as input. It traces the transformed program with the crashing input as pre-constraints using *preconstraint_trace* instruction by instruction, and collects the path constraints returned by it in TC. In case a negated jump instruction is encountered, the inverted constraint is saved in CO. In the end, the satisfiability of constraints in CO is checked, if it is unsatisfiable, the input is identified as a false positive, otherwise the constraints collected in CO can be used to generate input to reproduce the bug in the original program.

Note that, to err on the side of not overwhelming human analysts with false detections, the Crash Analyzer errs on the side of introducing false negatives over allowing false positives. That is, it is possible that detections marked as false positives by the Crash Analyzer, because they could not be directly reproduced in a symbolic trace, do actually represent bugs in the program. This will be discussed in detail in the case study shown in Section 2.3.5. Further improvements to the Crash Analyzer, beyond transformation-aware symbolic tracing, would improve T-FUZZ's effectiveness.

2.3 T-FUZZ Evaluation

We have implemented our prototype in Python based on a set of open source tools: the Fuzzer component was built on AFL [135], the Program Transformer was implemented

Algorithm 5: Process to filter out false positives

Input: $transformed\_program$: the transformed program
Input: c_addrs: addresses of negated conditional jumps
Input: $input$: the crashing input
Input: CA: the crashing address

$PC \leftarrow make\_constraint(input)$
$CT \leftarrow PC$
$CO \leftarrow \emptyset$
$TC, addr \leftarrow preconstraint\_trace(transformed\_program, CT, entry)$
while $addr \neq CA$ **do**
    if $addr \in c\_addrs$ **then**
        $CO \leftarrow CO \cup \neg TC$
    else
        $CO \leftarrow CO \cup TC$
    $TC, addr \leftarrow preconstraint\_trace(transformed\_program, CT, \mathrm{i})$
$CO \leftarrow CO \cup extract\_crashing\_condition(TC)$
$result \leftarrow SAT(CO)$

Output: $result$: A boolean indicating whether $input$ is a false positive
Output: CO: The set of constraints for generating the inputs in the original program

using the angr tracer [35] and radare2 [114], and the Crash Analyzer was implemented using angr [103].

To determine T-FUZZ's bug finding effectiveness, we performed a large-scale evaluation on three datasets (the DARPA CGC dataset, the LAVA-M dataset, and a set of 4 real-world programs built on wide-spread libraries, consisting of `pngfix/libpng`, `tiffinfo/libtiff`, `magick/ImageMagick`, and `pdftohtml/libpoppler`) and compared T-FUZZ against a set of state-of-art fuzzing tools.

The experiments were run on a cluster in which each node is running Ubuntu 16.04 LTS and equipped with an Intel i7-6700K processor and 32 GB of memory.

2.3.1 DARPA CGC Dataset

The DARPA CGC dataset [67] contains a set of vulnerable programs used in the Cyber Grand Challenge event hosted by DARPA. These programs have a wide range of authors,

functionalities, and vulnerabilities. Crucially, they also provide *ground truth* for the (known) vulnerabilities in these programs. The dataset contains a total of 248 challenges with 296 binary programs, as some of the challenges include multiple vulnerable binaries. For each bug in these programs, the dataset contains a set of inputs as a *Proof of Vulnerability*. These inputs are used as ground truth in our evaluation. Programs in this dataset contain a wide range of sanity checks on the input that are representative of real world programs, and thus are used widely as benchmark in related work [66, 96, 106].

In this evaluation, we run the CGC binaries with three different configurations: (1) to evaluate T-FUZZ against heuristic based approaches, we run AFL on the set of CGC binaries ; (2) to evaluate T-FUZZ against symbolic execution-based approaches, we run Driller [106] on the CGC binaries; (3) we run T-FUZZ on the same set of binaries. Each binary is fuzzed for 24 hours with an initial seed of "fuzz".

AFL. In this experiment, each binary is assigned one CPU core. Before fuzzing starts, we create a dictionary using angr [103] to help AFL figure out the possible magic values used in the program.

Driller. When running the experiment with Driller [52], each binary is assigned one CPU core for fuzzing and one CPU core for dedicated concolic execution (i.e., Driller uses two CPU cores, double the resources of the other experiments). For resource limits, we use the same settings as the original Driller evaluation [106].

T-Fuzz. To evaluate T-FUZZ, we assign the same CPU limits as for AFL and use one CPU core (half the resources of the Driller experiment). When multiple transformed binaries are generated, they are queued and fuzzed in first-in-first-out order.

To get an idea of the overall effectiveness of the system, we evaluate the three different configurations and discuss the bugs they found.

To validate the T-FUZZ results (i.e., the stability of T-FUZZ's program transformation strategy), we performed the DARPA CGC evaluation three times and verified that the same set of vulnerabilities was found each time. This makes sense, as aside from randomness in the fuzzing process, the T-Fuzz algorithm is fully deterministic.

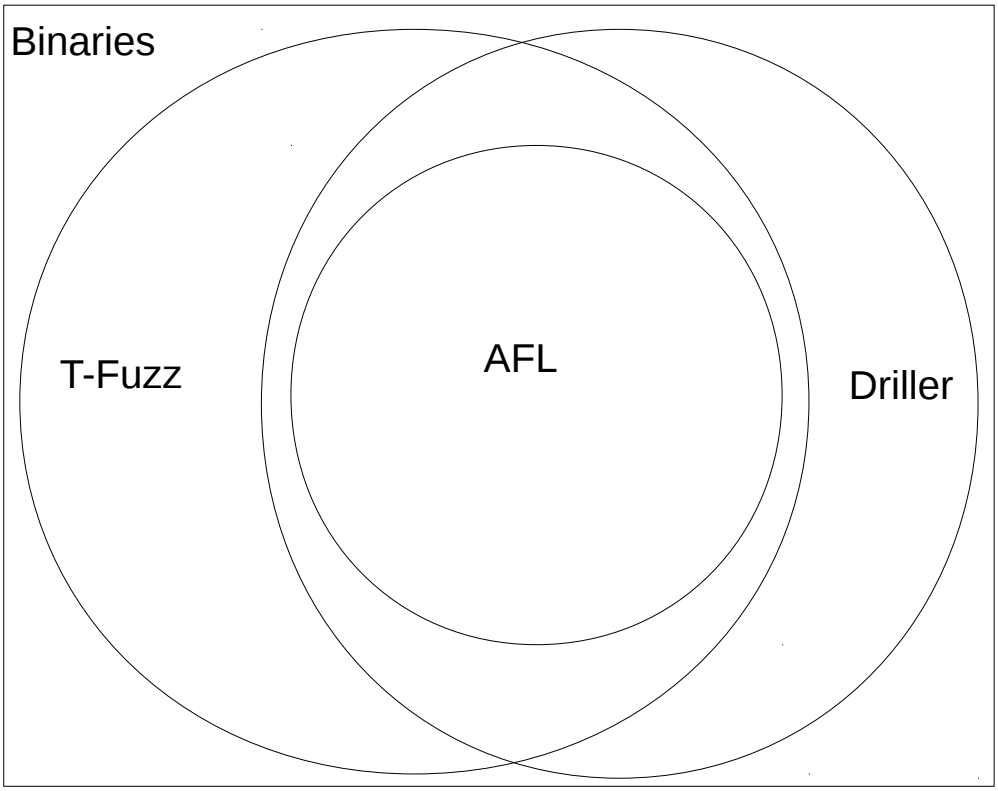


Figure 2.7. Venn diagram of bug finding results

Comparison with AFL and Driller

As the results in Figure 2.7 and Table 2.1 show, T-FUZZ *significantly* outperforms Driller in terms of bug finding. Given the time budget and resource limits mentioned above, T-FUZZ found bugs in 166 binaries (out of 296), compared to Driller which only found bugs in 121 binaries and AFL which found bugs in 105 binaries. All of the bugs found by AFL were discovered by both Driller and T-FUZZ. T-FUZZ found bugs in 45 additional binaries in which Driller did not, while failing to find bugs in 10 that Driller managed to crash. It is important to note that all the bugs found by T-FUZZ mentioned here are *true positives*, verified using the ground truth that the CGC dataset provides. The false positives resulting from T-FUZZ's analysis are discussed later.

Out of these 166 binaries in which T-FUZZ found crashes, 45 contain complex sanity checks that are hard for constraint solvers to generate input for. Failing to get accurate input that is able to bypass the "hard" sanity checks from a symbolic execution engine,

Table 2.1. Details of experimental results

Method	**Number of Binaries**
AFL	105
Driller	121
T-Fuzz	166
Driller$\setminus$AFL	16
T-Fuzz$\setminus$AFL	61
Driller$\setminus$T-Fuzz	10
T-Fuzz$\setminus$Driller	45

the fuzzing engine (AFL) in Driller keeps generating random inputs blindly until it uses up its time budget without making any progress in finding new code paths. This is where the difference between Driller and T-FUZZ comes in. Once the fuzzer gets "stuck", T-FUZZ disables the offending conditional checks, and lets the fuzzer generate inputs to cover code previously protected by them, finding new bug candidates. We use a case study in Section 2.3.5 to demonstrate the difference in detail.

Driller found bugs in 10 binaries for which T-FUZZ failed to find (true) bugs. This discrepancy is caused by 2 limitations in the current implementation of T-FUZZ. First, if a crash caused by a false positive stemming from an NCC negation occurs in the code path leading to a true bug, the execution of the transformed program will terminate with a crash without executing the vulnerable code where the true bug resides (L1). T-FUZZ "lost" to Driller in three of the 10 binaries because of this limitation. Secondly, when the true bug is hidden deep in a code path containing many sanity checks, T-FUZZ undergoes a sort of "transformation explosion", needing to fuzz too many different versions of transformed program to trigger the true bug (L2). While this is not very frequent in our experiments, it does happen: T-FUZZ failed to find the bugs in the remaining 7 binaries within the 24-hour time budget. We plan to explore these issues in T-FUZZ in our future work.

These results show that T-FUZZ greatly improves the performance of bug finding via fuzzing. By disabling the sanity checks, T-FUZZ finds bugs in 45 more CGC binaries than Driller. The additional bugs found by T-FUZZ are heavily guarded by hard checks and hidden in deep code paths of programs.

Of course, there is no requirement to use only a single analysis when performing fuzzing in the real world. The union of detections from T-FUZZ and Driller is *176* identified bugs, *significantly* higher than any other reported metric from experimentation done on the CGC dataset.

Comparison with other tools

In Steelix [66], 8 binaries of the CGC dataset were evaluated. Steelix only found an additional bug in KPACA_00001. As mentioned in the Steelix paper, the main challenge for

fuzzing it is to bypass the check for magic values inside the program. Using a manually-provided seed that reaches the sanity check code in the program, Steelix detects the comparison against magic values in the program and successfully generates input that bypassed the sanity check, finding the bug in less than 10 minutes. T-FUZZ finds the bug in its first transformed program in around 15 minutes – without requiring manual effort to provide a seed that reaches the checking code.

In VUzzer [96], 63 of the CGC binaries were evaluated with a 6-hour time budget, among which VUzzer found bugs in 29. T-FUZZ found bugs in 47 of the 63 binaries, 29 of which were found within 6 hours. In the 29 binaries in which VUzzer found bugs, T-FUZZ found bugs in 23 of them (in 6 hours). T-FUZZ failed to find the bugs within a 24-hour time budget in the remaining 6 for the same reasons mentioned above (2 for L1 and 4 for L2). However, VUzzer is unable to run on the full CGC dataset, making a comprehensive comparison difficult.

2.3.2 LAVA-M Dataset

The LAVA dataset contains a set of vulnerable programs created by automatically injecting bugs using the technique proposed in [27]. **LAVA-M** is a subset of the LAVA dataset consisting of 4 utilities from coreutils, each of which contains multiple injected bugs. The authors evaluated a coverage-guided fuzzing tool (FUZZER) and a symbolic analysis based tool (SES) for 5 hours [27]. The dataset was also used in VUzzer [96] and Steelix [66] as part of their evaluation. As at the time of this evaluation, Steelix is not available and VUzzer cannot be run (due to dependencies on a closed IDA plugin), we ran T-FUZZ for 5 hours on each of the binaries in LAVA-M dataset to compare our results with those stated by the authors of VUzzer and Steelix in their papers.

The evaluation results are summarized in Table 2.2. The results of T-FUZZ is shown in the last column and results from other work are shown in other columns. It is important to note that the bugs mentioned here are bugs that have been confirmed by Crash Analyzer or manually. We first run Crash Analyzer on the reported bugs and then manually analyzed the ones marked as false positives by Crash Analyzer. From the results we can see that

T-FUZZ found almost the same number of bugs as Steelix in `base64` and `uniq`, far more bugs in `md5sum` and less bugs in `who` than Steelix. The reasons are summarized as follows:

- Steelix and VUzzer performed promisingly well on `base64`, `uniq` and `who` because of two important facts. First of all, bugs injected into these programs are all protected by sanity checks on values copied from input against magic bytes **hardcoded** in the program. Thus, the static analysis tools used in Steelix or VUzzer can easily recover the expected values used in the sanity checks that guard the injected bugs. Secondly, the LAVA-M dataset provides well-formated seeds that help the fuzzer reach the code paths containing injected bugs. If either of the conditions fails to hold, Steelix and VUzzer would perform worse. However, with the inability to evaluate Steelix (which is unavailable) and VUzzer (which we could not run), this is hard to verify.

- T-FUZZ can trigger the bugs in code paths protected by "hard" checks which both Steelix and VUzzer can not bypass. In md5sum, T-FUZZ found 15 bugs that were not found by Steelix. These bugs are protected by sanity checks on values computed from the MD5 sum of specified files, instead of being copied from the input directly. As the expected values can no longer be constructed easily using the hardcoded values present in the program, Steelix failed to find these bugs.

- T-FUZZ performed worse than Steelix in `who` due to the limited time budget. As the number of injected bugs is huge and each injected bug is protected by a sanity check, T-FUZZ was limited by the emergent issue of "transformal explosion" and generated 578 transformed programs. Within 5 hours, T-FUZZ only found 63 bugs.

In summary, T-FUZZ performs well in comparison with state-of-art fuzzing tools in terms of bug finding even given conditions favorable for other counterparts. In the presence of "hard" checks on the input, e.g. checksums, T-FUZZ performs better than existing techniques for finding bugs "guarded" by such checks.

Table 2.2. LAVA-M Dataset evaluation results

program	Total # of bugs	FUZZER	SES	VUzzer	Steelix	T-FUZZ
base64	44	7	9	17	43	43
md5sum	57	2	0	1	28	49
uniq	28	7	0	27	24	26
who	2136	0	18	50	194	63

2.3.3 Real-world Programs

We evaluated T-FUZZ on a set of real-world program/library pairs (`pngfix/libpng`, `tiffinfo/libtiff`, `magick/libMagickCore`, and `pdftohtml/libpoppler`) and compare it against AFL in terms of crashes found. Each program was fuzzed for 24 hours with random seeds of 32 bytes.

Table 2.3 summarizes the number of unique true-positive crashes found by T-FUZZ and AFL. T-FUZZ found 11, 124, 2 and 1 unique crashes in `pngfix`, `tiffinfo`, `magick` and `pdftohtml` respectively. AFL did not trigger any crashes in `pngfix`, `magick` and `pdftohtml` and only found less than half of the crashes T-FUZZ found in `tiffinfo`. As random seeds were provided, AFL got stuck very quickly, failing to bypass the sanity checks on the file type bytes (the file header for PNG files, the "II" bytes for TIF files, etc.). Within 24 hours, although AFL succeeded in generating inputs to bypass these checks, it failed to generate inputs that could bypass further sanity checks in the code, thus being unable to find bugs protected by them. In particular, in `pngfix`, `magick`, and `pdftohtml`, the bugs found by T-FUZZ are hidden in code paths protected by multiple sanity checks, and thus were not found by AFL; in `tiffinfo`, AFL found crashes, failing to find the 71 additional crashes caused by code paths that are guarded by more sanity checks.

These larger real-world programs demonstrate drawbacks of the underlying symbolic execution engine: angr simply does not have the environment support to scale to these programs. While this is not something that we can fix in the prototype without extensive effort by the angr team itself, our observation is that T-Fuzz actually causes surprisingly few false positives in practice. For example, for `pdftohtml`, the true positive bug was the only alert that T-Fuzz generated.

After inspecting the crashes resulting from T-Fuzz, we found 3 new bugs (marked by * in Table 2.3): two in `magick` and one in `pdftohtml`. It is important to note that these bugs are present in the latest stable releases of these programs, which have been intensively tested by developers and security researchers. One of the new bugs found in `magick` has already been fixed in a recent revision, and we have reported the remaining 2 to the developers [15, 110], waiting to be acknowledged and fixed. Importantly, AFL failed to trigger any of these 3 bugs. As these bugs are hidden in code paths protected by several checks, it is very hard for AFL to generate inputs bypassing all of them. In constrast, T-FUZZ successfully found them by disabling checks that prevented the fuzzer-generated input to cover them.

Table 2.3. Real-world programs evaluation results, with crashes representing new bugs found by T-FUZZ in `magick` (2 new bugs) and `pdftohtml` (1 new bug) and crashes representing previously-known bugs in `pngfix` and `tiffinfo`.

Program	AFL	T-FUZZ
pngfix + libpng (1.7.0)	0	11
tiffinfo + libtiff (3.8.2)	53	124
magick + ImageMagick (7.0.7)	0	2*
pdftohtml + libpoppler (0.62.0)	0	1*

2.3.4 False Positive Reduction

T-FUZZ utilizes a Crash Analyzer component to filter out false positive detections stemming from program transformations. This component is designed to avoid the situation, common in related fields such as static analysis, where a vulnerability detection tool overwhelms an analyst with false positives. In this section, we explore the need for this tool, in terms of its impact on the alerts raised by T-FUZZ.

False-positive-prone static analyses report false positive rates of around 90% for the analysis of binary software [94, 103]. Surprisingly, we have found that even in the presence of program transformations that could introduce unexpected behavior, T-FUZZ produces relatively few false-positive bug detections. We present an analysis of the alerts raised by T-FUZZ on a sampling of the CGC dataset and LAVA-M dataset in Table 2.4 and Table 2.5, along with the reports from our Crash Analyzer and ratio of false negatives.

In the CGC dataset, T-FUZZ provides the Crash Analyzer component with 2.8 alerts for every true positive detection on average, with a median of 2 alerts for every true positive. In the LAVA-M dataset, T-FUZZ only raised Crash Analyzer with 1.1 alerts for each true bug on average. Compared to static techniques, this is a significant advantage — even without the Crash Analyzer component, a human analyst would have to investigate only three alerts to locate an actual bug. Compared with other fuzzing techniques, even with the aggressive false positive reduction performed by the Crash Analyzer (resulting in only actual bug reports as the output of T-FUZZ), T-FUZZ maintains higher performance than other state-of-the-art systems.

As mentioned in Section 2.2.4, the Crash Analyzer may mark as false positives detections that actually *do* hint at a bug (but are not trivially repairable with the adopted approach), resulting false negative reports. E.g., if there is a "hard" check (e.g., checksum) that was disabled in the code path leading to a crash found by T-FUZZ, applying Crash Analyzer on the crash would involve solving hard constraint sets. As current constraint solvers can not determine the SATness of such hard constraint sets, Crash Analyzer would err on the false negative side and mark it as a false bug. Another example is shown in Section 2.3.5. In the selected sample of CGC dataset shown in Table 2.4, Crash Analyzer mark detected crash in the first 3 binaries as false alerts. In LAVA-M dataset, Crash Analyzer has an average false negative rate of 15%. It shows a slightly higher false negative rate (30%) in `md5sum` because 15 of the detected crashes are in code paths protected by checks on MD5 checksums.

2.3.5 Case Study

CROMU_00030[2] contains a stack buffer overflow bug (line 11) in a code path guarded by multi-stage "hard" checks. As shown in Listing 2.4, to reach the buggy code, the input needs to bypass 10 rounds of checks and each round includes a basic sanity check (line 19), a check on checksum (line 25) and a check on the request (line 30, in *handle_packet*).

When T-FUZZ fuzzes this binary, after roughly 1 hour of regular fuzzing, the fuzzer gets "stuck", failing to pass the check in line 25. T-FUZZ stops the fuzzer and uses the

[2]↑This program simulates a game over a protocol similar to IEEE802.11 and is representative of network programs.

Table 2.4. A sampling of T-FUZZ bug detections in CGC dataset, along with the amount of false positive alerts that were filtered out by its Crash Analyzer.

Binary	# Alerts	# True Alerts	# Reported Alerts	% FN
CROMU_00002	1	1	0	100%
CROMU_00030	1	1	0	100%
KPRCA_00002	1	1	0	100%
CROMU_00057	2	1	1	0
CROMU_00092	2	1	1	0
KPRCA_00001	2	1	1	0
KPRCA_00042	2	1	1	0
KPRCA_00045	2	1	1	0
CROMU_00073	3	1	1	0
KPRCA_00039	3	1	1	0
CROMU_00083	4	1	1	0
KPRCA_00014	4	1	1	0
KPRCA_00034	4	1	1	0
CROMU_00038	5	1	1	0
KPRCA_00003	6	1	1	0

Table 2.5. T-FUZZ bug detections in LAVA-M dataset, along with the amount of false positive alerts that were filtered out by its Crash Analyzer.

Program	# Alerts	# True Alerts	# Reported Alerts	% FN
base64	47	43	40	6%
md5sum	55	49	34	30%
uniq	29	26	23	11%
who	70	63	55	12%

```c
int main() {
  int step = 0;
  Packet packet;
  while (1) {
    memset(packet, 0, sizeof(packet));
    if (step >= 9) {
       char name[5];
       // stack buffer overflow BUG
       int len = read(stdin, name, 25);
       printf("Well done, %s\n", name);
       return SUCCESS;
    }
    // read a packet from the user
    read(stdin, &packet, sizeof(packet));
    // initial sanity check
    if(strcmp((char *)&packet, "1212") == 0) {
      return FAIL;
    }
    // other trivial checks on the  packet omitted
    if (compute_checksum(&packet) != packet.checksum) {
      return FAIL;
    }
    // handle the request from the user, e.g., authentication
    if (handle_packet(&packet) != 0) {
      return FAIL;
    }
    // all tests in this step passed
    step ++;
  }
}
```

Listing 2.4. Code excerpt of CROMU_00030, slightly simplified for readability

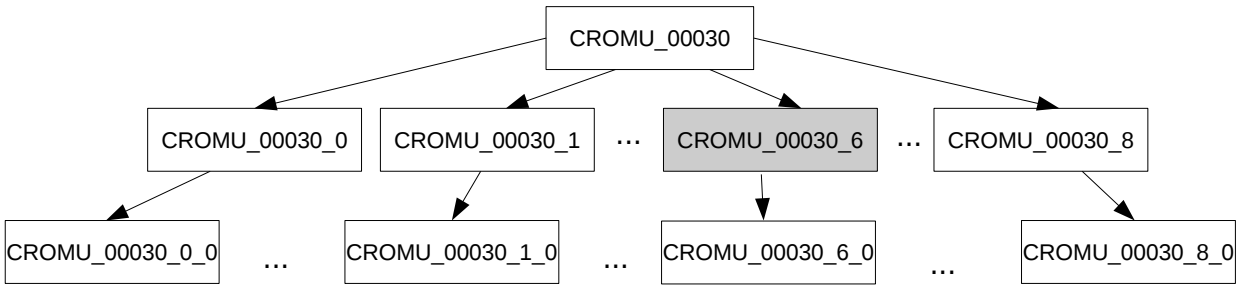


Figure 2.8. The transformed binaries T-FUZZ generates and fuzzes

fuzzer-generated inputs to detect NCC candidates, pruning undesired candidates using the algorithm from Section 2.2.1, returning a set of 9 NCC candidates. Next T-FUZZ transforms the original program and generates 9 different binaries (shown as CROMU_00030_0-CROMU_00030_8 in Figure 2.8) with one detected NCC candidate removed in each. They are then fuzzed and transformed sequentially in FIFO order in the same way as the original.

When CROMU_00030_6 (marked as grey in Figure 2.8), which is the binary with the check in line 8 negated, is fuzzed, a crash is triggered within 2 minutes, which is the true bug in the original binary. The total time it takes T-FUZZ to find the bug is about 4 hours, including the time used for fuzzing the original binary (CROMU_00030) and the time for fuzzing CROMU_00030_0-CROMU_00030_5. After the real bug is found by fuzzing CROMU_00030_6, T-FUZZ continues to fuzz and transform other (transformed) binaries until it uses up its time budget.

It is important to note that T-FUZZ can also find the bug by fuzzing the transformed binary with the sanity checks in line 25 and 30 negated. In that case, all the user provided input "bypasses" these two complex checks and the buggy code in line 11 is executed after looping for 10 iterations. As we fuzz the transformed binaries in FIFO order, this configuration is not reached within the first 24 hours.

In contrast, Driller failed to find the bug in this binary. Driller's symbolic execution engine cannot produce an accurate input to bypass the check in line 25, as it is too complex. Unable to get inputs to guide execution through the sanity check, the fuzzer blindly mutates the inputs without finding any new paths until it uses up its time budget. Note also that it

is highly unlikely for Driller to generate input to bypass the check in line 30 even without the check in line 25 because of the complexity involved in encoding the state of the protocol.

Listing 2.4 also showcases an example where Crash Analyzer marks a true bug as a false positive. As the *step* variable is not read from user and initialized as 0 in line 2, when the Crash Analyzer reaches the crash in CROMU_00030_6, the accumulated constraints set is $\{step == 0, step >= 9\}$ which is UNSAT, thus it is marked as false positive. This bug was identified by manual analysis.

2.4 Related Work

2.4.1 Feedback Based Approaches

Feedback based approaches make heuristics of possible magic values and their positions in the input based on feedback from the target program. E.g., AFL [135] and libFuzzer [111] can automatically guess syntax tokens based on the change in coverage and mutate input based on those tokens [134]. Further, AFL-lafintel [1] and improve the feedback by dividing a check on multiple bytes values into multiple nested checks on one byte values and Steelix [66] introduces "comparison progress" of checks to the feedback.

These approaches are based on *hindsight* to extend coverage past a check, i.e., the fuzzer must have already generated/mutated an input that passes some check in the program. Also, these approaches cannot handle checks on values computed on the fly or based on other input values such as checksums. T-FUZZ, on the other hand, is not limited by such restrictions

2.4.2 Symbolic and Concolic Execution Based Approaches

Symbolic execution encodes the sanity checks along a program path as a set of constraints (represented as logical formula), reducing the problem of finding a passing input to solving the encoded constraints. Several tools have implemented symbolic execution, e.g., KLEE [16], Veritesting [6], SAGE [38], DART [37], SYMFUZZ [18], CUTE [102], SmartFuzz [74], and Driller [106], covering domains like automatic testcase generation, automatic bug finding and fuzzing.

Among the tools mentioned above, Driller [106] is the closest to T-FUZZ. Driller uses selective concolic execution to generate inputs when the fuzzer gets "stuck". As mentioned in previous sections, symbolic and concolic execution based approaches, including Driller, suffer in scalability and ability to cover code paths protected by "hard" checks, where T-FUZZ excels.

2.4.3 Taint Analysis Based Approaches

Dynamic taint analysis identifies the dependencies between the program logic and input. Taintscope [128] focuses mutating the security-sensitive parts of the input identified by dynamic taint analysis. Other works apply additional analysis based on the identified dependencies to improve input generation, e.g., VUzzer uses data-flow and control analysis, Dowser [53] and BORG [76] use symbolic analysis.

Dynamic taint analysis is heavy weight, application of other techniques (data and control flow analysis in VUzzer and symbolic analysis in Dowser and BORG) adds up the overhead. In contrast,T-FUZZ only uses lightweight dynamic tracing technique for identifying and disabling sanity checks in the target program.

2.4.4 Learning Based Approaches

This category of approaches generate inputs by learning from large amount of valid inputs. E.g., Skyfire [127] and Learn&Fuzz [39] generate seeds or inputs for the fuzzer using the learned probabilistic distribution of different values from samples, while GLADE [10] generates inputs based on the synthesized grammar learned from provided seeds.

Learning based approaches has been shown to be effective in generating well structured inputs (e.g., XML files) for fuzzers in Skyfire [127] and Learn&Fuzz [39]. However, it is difficult to learn less structured inputs like images or complex dependencies among different parts of data like checksums without external knowledge. In addition, learning requires a large corpus of valid inputs as training set. In contrast, T-FUZZ does not have such limitations.

2.4.5 Program Transformation Based Approaches

Some existing work uses the idea of program transformation to overcome sanity checks in the target program, but requires significant **manual** effort. E.g., Flayer [28] relies on user provided addresses of the sanity checks to perform transformation. TaintScope [128] depends on a on a pair of inputs with one able to bypass a sanity check on checksum and the other not, requiring a significant amount of manual analysis. MutaGen [58] depends on the availability and identification of program code that can generate the proper protocols, a process involving much manual effort. In addition, they use dynamic instrumentation to alter the execution of the target, which typically involves a slowdown of 10x in execution speed.

T-FUZZ is the only program transformation-based technique, known to us, that is able to leverage program transformation, in a completely automated way, to augment the effectiveness of fuzzing techniques.

2.5 Summary

Mutational fuzzing so far has been limited to producing new program inputs. Unfortunately, hard checks in programs are almost impossible to bypass for a mutational fuzzer (or symbolic execution engine). Our proposed technique, transformational fuzzing, extends the notion of mutational fuzzing to the program as well, mutating both program and input.

In our prototype implementation *T-FUZZ* we detect whenever a baseline mutational fuzzer (AFL in our implementation) gets stuck and no longer produces inputs that extend coverage. Our lightweight tracing engine then infers all checks that could not be passed by the fuzzer and generates mutated programs where the checks are negated. This change allows our fuzzer to produce input that trigger deep program paths and therefore find vulnerabilities hidden deep in the program itself.

We have evaluated our prototype on the CGC dataset, the LAVA-M dataset, and 4 real-world programs (`pngfix`, `tiffinfo`, `magick` and `pdftohtml`). In the CGC dataset, T-FUZZ finds true bugs in 166 binaries, improving the results by 45 binaries compared to Driller and 61 binaries compared to AFL alone. In the LAVA-M dataset, T-FUZZ shows significantly

better performance in the presence of "hard" checks in the target program. In addition, we have found 3 new bugs in evaluating real-world programs: two in `magick` and one in `pdftohtml`. T-FUZZ is available at https://github.com/HexHive/T-Fuzz.

2.6 Future Work

As our evaluation shows in Section 2.3.1, T-FUZZ is currently limited by false crashes (L1) and transformation explosions (L2). We plan to improve T-FUZZ by developing better heuristics to transform target programs. In addition, the underlying symbolic execution engine, angr, lacks support for environmental modeling, and is not scalable as it needs to save an exponential number of program states. Therefore, the CrashAnalyzer component does not support large real-world programs. We plan to improve angr by adding the necessary modeling support required by the C library, and improve its symbolic tracing algorithm by removing the unneeded program states while following a dynamic trace.

3. USBFUZZ: FUZZING USB DRIVERS BY DEVICE EMULATION

3.1 Background

The USB architecture implements a complex but flexible communication protocol that has different security risks when hosts communicate with untrusted devices. Fuzzing is a common technique to find security vulnerabilities in software, but existing state-of-the-art fuzzers are not geared towards finding flaws in drivers of peripheral devices.

3.1.1 USB Architecture

Universal Serial Bus (USB) was introduced as an industry standard to connect commodity computing devices and their peripheral devices. Since its inception, several generations of the USB standard (1.x, 2.0, 3.x) have been implemented with increasing bandwidth to accommodate a wider range of applications. There are over 10,000 different USB devices [115].

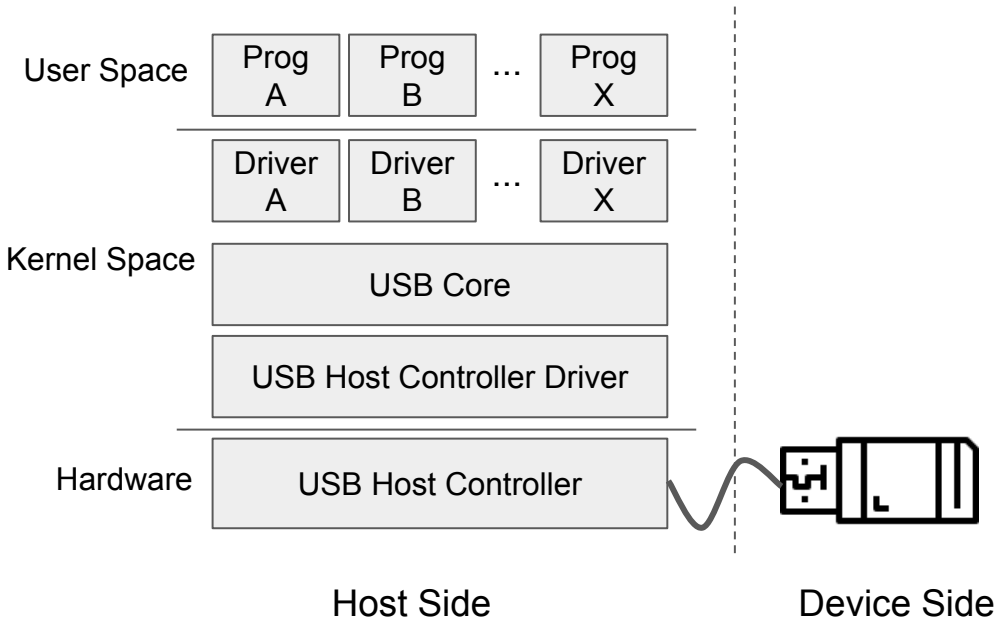


Figure 3.1. USB architecture

USB follows a master-slave architecture, divided into a single host side and potentially many device sides. The device side acts as the slave, and implements its own functionality.

The host side, conversely, acts as the master, and manages every device connected to it. All data communication must be initiated by the host, and devices are not permitted to transmit data unless requested by the host.

The most prominent feature of the USB architecture is that it allows a single host to manage different types of devices. The USB standard defines a set of requests that every USB device must respond to, among which the most important are the device descriptor (containing the vendor and product IDs) and the configuration descriptor (containing the device's functionality definition and communication requirements), so that the host-side software can use different drivers to serve different devices according to these descriptors.

The host side adopts a layered architecture with a hardware-based host controller (see Figure 3.1). The host controller provides physical interfaces (using a root hub component), and supports multiplexing device access, and the host controller driver provides a hardware-independent abstraction layer for accessing the physical interfaces. The *USB core* layer, built on top of the host controller driver, is responsible for choosing appropriate drivers for connected devices and provides core routines to communicate with USB devices. Drivers for individual USB devices (located on top of the USB core) first initialize the device based on the provided descriptors, then interface with other subsystems of the host OS. Userspace programs use APIs provided by various kernel subsystems to communicate with the USB devices.

USB drivers consist of two parts: (i) probe routine to initialize the driver and (ii) function routines to interface with other subsystems (e.g, sound, network, or storage) and deregister the driver when the device is unplugged. Existing USB fuzzers focus exclusively on the probe routines, ignoring other function routines, because probe functions are invoked automatically when the device is plugged in, while other function routines are usually driven by userspace programs.

3.1.2 USB Security Risks

USB exposes kernel access from externally-connected peripherals, and therefore poses an attack surface. In the past years, several USB-based attacks have been devised to compromise the security of a computer system. We classify the existing USB-based attacks below.

C1. Attacks on implicit trust. As a hardware interface, both OSes and the USB standard implicitly assume that the device is trustworthy. A wide range of USB-based attacks [21, 77, 120] reprogram the device firmware. The reprogrammed devices look like regular USB thumb drives, but perform additional tasks like keylogging (BadUSB [61]) or injecting keystrokes and mouse movements, thus allowing installation of malware, exfiltrating sensitive information (USB Rubber Ducky [13]), installing backdoors, or overriding DNS settings (USBDriveby [57]).

C2. Electrical attacks. Here, the attacker uses the power bus in the USB cable to send a high voltage to the host, causing physical damage to the hardware components of the host computer. USBKiller [123] is the best known attack falling into this category.

C3. Attacks on software vulnerabilities. The attacker leverages a vulnerability in the USB stack or device drivers. As an example, Listing 3.1 highlights a Linux kernel vulnerability reported in CVE-2016-2384 [78] where a malicious USB-MIDI [4] device with incorrect endpoints can trigger a double-free bug (one in line 7, and the other in line 18 when the containing object (`chip->card`) is freed).

Memory bugs similar to Listing 3.1 can be disastrous and may allow an adversary to gain control of the host system, because device drivers run in privileged mode (either in the kernel space or as a privileged process). An exploit for the above vulnerability allows full adversary-controlled code execution [62]. Since devices connected to USB may function as any arbitrary device from the perspective of the host system, the USB interface exposes attacker-controlled input to any service or subsystem of the kernel that is connected through a USB driver. Similar exploits target the storage system of Windows [65].

```
// in snd_usbmidi_create
if (quirk && quirk->type == QUIRK_MIDI_MIDIMAN)
  err = snd_usbmidi_create_endpoints_midiman(umidi, &endpoints[0]);
else
  err = snd_usbmidi_create_endpoints(umidi, endpoints);
if (err < 0) {
  snd_usbmidi_free(umidi);
  return err;
}
// in usb_audio_probe, snd_usb_create_quirk calls snd_usbmidi_create
err = snd_usb_create_quirk(chip, intf, &usb_audio_driver, quirk);
if (err < 0)
  goto __error;
//...
__error:
  if (chip)
    if (!chip->num\_interfaces)
      snd_card_free(chip->card);
```

Listing 3.1. CVE-2016-2384 [78] vulnerability

These security risks are rooted in a basic assumption: hardware is difficult to modify and can be trusted. On one hand, as USB connects hardware devices to computer systems, security issues were neither part of the design of the USB standard nor host side software implementation, making attacks on the trust model (C1) and electrical attacks (C2) possible. On the other hand, device driver developers tend to make assumptions regarding the data read from the device side, e.g., the descriptors are always legitimate. This assumption results in the problem that unexpected data read from the device side may be improperly handled. Even if the developers try to handle unexpected values, as recently disclosed bugs demonstrate [43], code is often not well tested due to the difficulty in providing exhaustive unexpected data during development.[1] In other words, when a device driver is written, the programmer can speculate about unexpected inputs, but it is infeasible to create arbitrary hardware that provides such faulty inputs. This results in poorly-tested error-handling code paths.

However, recent research has fundamentally changed this basic assumption. Some USB device firmware is vulnerable, allowing attackers to control the device and messages sent on

[1]↑Special hardware that provides unexpected data from the USB device side exists (e.g., Ellisys USB Explorer [31]), however it is either not used because of its cost, or the drivers are not sufficiently tested.

the bus. In addition, with the adoption of recent technologies such as Wireless USB [130] and USBIP [90], the USB interface is exposed to networked devices, turning USB-based attacks into much easier network attacks. Finally, reprogrammable USB devices (e.g., FaceDancer [41]) allow the implementation of arbitrary USB devices in software.

3.1.3 Fuzzing the USB Interface

Given the security risks, there have been several fuzzing tools targeting the USB interface. This section briefly analyzes these existing fuzzing tools and serves to motivate our work.

The first generation of USB fuzzers targets the device level. vUSBf [100] uses a networked USB interface (usbredir [86]), and umap2 [51] uses programmable hardware (FaceDancer [41]) to inject random hardware input into the host USB stack. Though easily portable to other OSes, they are dumb fuzzers and cannot leverage coverage information to guide their input mutation, rendering them inefficient.

The recent usb-fuzzer [42] (an extension of the kernel fuzzer syzkaller [47]) injects fuzz inputs into the IO stack of the Linux kernel using a custom software-implemented host controller combined with a coverage-guided fuzzing technique. The adoption of coverage-guided fuzzing has led to the discovery of many bugs in the USB stack of the Linux kernel [42]. However, usb-fuzzer is tightly coupled with the Linux kernel, making it hard to port to other OSes.

All existing USB fuzzers focus exclusively on the probe routines of drivers, not supporting fuzzing of the remaining function routines. The status-quo of existing USB fuzzers motivates us to build a flexible and modular USB fuzzing framework that is portable to different environments and easily customizable to apply coverage-guided fuzzing or dumb fuzzing (in kernels where coverage collection is not yet supported), and allows fuzzing a broad range of probe routines or focusing on the function routines of a specific driver.

3.2 Threat Model

Our threat model consists of an adversary that attacks a computer system through the USB interface, leveraging a software vulnerability in the host software stack to achieve goals

such as privilege escalation, code execution, or denial of service. Attacks are launched by sending prepared byte sequences over the USB bus, either attaching a malicious USB device to a physical USB interface or hijacking a connection to a networked USB interface (e.g., in USBIP [90] or usbredir [86]).

3.3 USBFUZZ Design

Device drivers handle inputs both from the device side and from the kernel. The kernel is generally trusted but the device may provide malicious inputs. The goal of USBFUZZ is to find bugs in USB drivers by repeatedly testing them using random inputs generated by our fuzzer, instead of the input read from the device side. The key challenge is how to feed the fuzzer generated inputs to the driver code. Before presenting our approach, we discuss the existing approaches along with their respective drawbacks.

Approach I: using dedicated hardware. A straight-forward solution is to use dedicated hardware which returns customizable data to drivers when requested. For USB devices, FaceDancer [41] is readily available and used by umap2 [51]. This approach follows the data paths in real hardware and thus covers the complete code paths and generates reproducible inputs. However, there are several drawbacks in such a hardware-based approach. First, dedicated hardware parts incur hardware costs. While $85 for a single FaceDancer is not prohibitively expensive, fuzzing campaigns often run on 10s to 1000s of cores, resulting in substantial hardware cost. Similarly, connecting physical devices to fuzzing clusters in a server center results in additional complexity. Second, hardware-based approaches do not scale as one device can only fuzz one target at a time. Hardware costs and lack of scalability together render this approach expensive. Finally, this approach is hard to automate as hardware operations (e.g., attaching and detaching a device to and from a target system) are required for each test iteration.

Approach II: data injection in IO stack. This approach modifies the kernel to inject fuzz data to drivers at a certain layer of the IO stack. For example, usb-fuzzer in syzkaller [47] injects fuzz data into the USB stack through a software host controller (`dummy`

`hcd`), replacing the driver for the hardware host controller. PeriScope [105] injects fuzzer generated input to drivers by modifying MMIO and DMA interfaces.

Compared to hardware-based approaches, this approach is cheap, scalable, and can be automated to accommodate fuzzing. However, this solution struggles with portability as its implementation is tightly coupled to a given kernel layer (and sometimes kernel version). In addition, it requires deep understanding of the hardware specification and its implementation in the kernel. As input is injected at a specific layer of the IO stack, it cannot test code paths end-to-end, and thus may miss bugs in untested code paths (as we show in Section 3.5.4).

Design Goals. After evaluating the above approaches, we present the following design goals:

G1. Low Cost: The solution should be cost-effective and hardware-independent.

G2. Portability: The solution should be portable to test other OS and platforms, avoiding deep coupling with a specific kernel version.

G3. Minimal Required Knowledge: The interaction between the driver, the USB device, and the rest of the system is complex and may be different from device to device. The solution should require minimal knowledge of the USB standard and the device.

USBFUZZ's approach. At a high-level, USBFUZZ leverages an emulated USB device to feed random input to device drivers. The target kernel (hosting the tested device drivers) runs in a virtual machine (VM) and the emulated USB device is integrated into the VM. The hypervisor in the VM transparently sends read/write requests from the drivers of the guest kernel to the emulated device (and not to real hardware) without any changes to the USB system in the target kernel. The emulated USB device, on the other hand, responds to kernel IO requests using the fuzzer-generated input, instead of following the specification of a device.

As a software-based solution, an emulated device does not incur any hardware cost and is highly scalable, as we can easily run multiple instances of a virtual machine to fuzz multiple instances of a target kernel in parallel, satisfying **G1**—low cost. Because our solution implements an emulated hardware device, it is decoupled from a specific kernel or version.

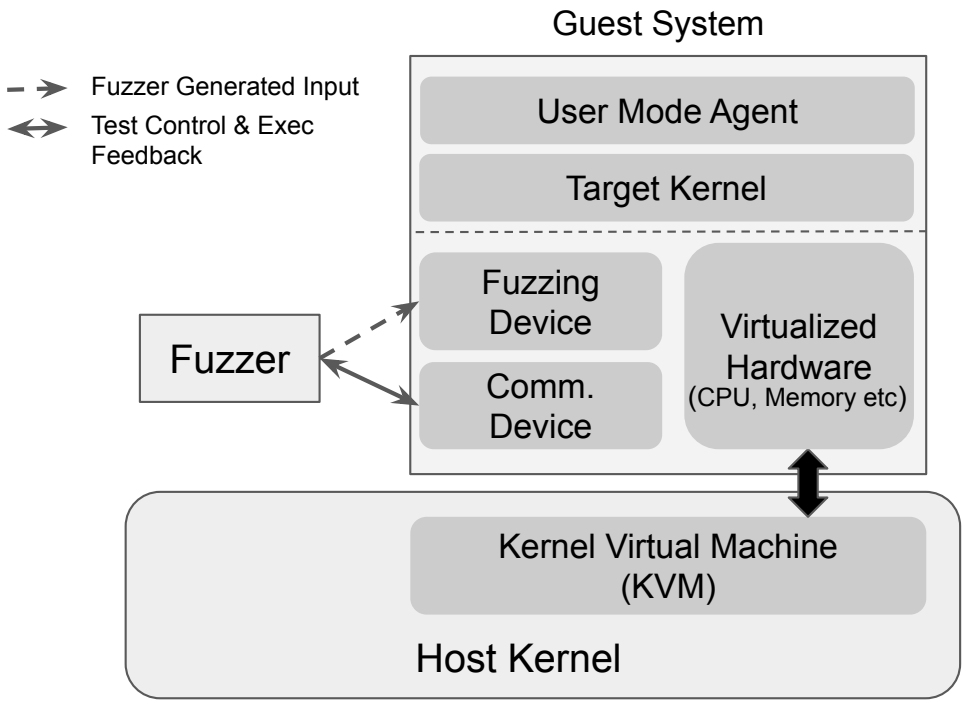


Figure 3.2. Overview of USBFUZZ

One implementation of the emulated device can be used to provide random input to device drivers running on different kernels on different platforms, satisfying **G2**—portability. As this solution works at the device level, no knowledge of the software layers in the kernel is required. In addition, based on mature emulators such as QEMU, a developer only needs to understand the data communication protocol, satisfying **G3**—minimal required knowledge.

Based on these goals, we designed USBFUZZ, a modular framework to fuzz USB device drivers. Figure 3.2 illustrates the overall design of USBFUZZ. The following list summarizes high level functionalities of its main components.

Fuzzer: The fuzzer runs as a userspace process on the host OS. This component performs the following tasks: (i) mutating the data fed to device drivers in the target kernel; and (ii) monitoring and controlling test execution.

Guest System: The guest system is a virtual machine that runs a target kernel containing the device drivers to test. It provides support for executing the guest code, emulating the fuzzing device as well as the supporting communication device.

Target Kernel: The target kernel contains the code (importantly, device drivers) and runs inside the guest system. The drivers in the kernel are tested when they process the data read from the emulated fuzzing device.

Fuzzing Device: The fuzzing device is an emulated USB device in the guest system. It is connected through the emulated USB interface to the guest system. However, instead of providing data according to the hardware specification, it forwards the fuzzer-generated data to the host when the target kernel performs IO operations on it (shown in Section 3.3.1).

Communication Device: The communication device is an emulated device in the guest system intended to facilitate communication between the guest system and the fuzzer component. It shares a memory region and provides synchronization channels between the fuzzer component and the guest system. The shared memory region also shares coverage information in coverage-guided fuzzing (shown in Section 3.3.2).

User Mode Agent: This userspace program runs as a daemon process in the guest system. It monitors the execution of tests (shown in Section 3.3.3). Optionally, it can be customized to perform additional operations on the fuzzing device to trigger function routines of drivers during focused fuzzing (demonstrated in Section 3.5.4).

The modular design of USBFUZZ, in combination with the emulated fuzzing device, allows fuzzing USB device drivers on different OSes and applying different fuzzing techniques with flexible configuration based on the target system, e.g., coverage-guided fuzzing to leverage feedback, or dumb fuzzing without any feedback to explore certain provided USB traces (dumb fuzzing is useful when coverage information is not available). In this work, we applied coverage-guided fuzzing to the Linux kernel (discussed in Section 3.3.4), and dumb fuzzing to FreeBSD, MacOS, and Windows using cross-pollination seeded by inputs generated from fuzzing Linux.

3.3.1 Providing Fuzzed Hardware Input

Our input generation component extends AFL, one of the most popular mutational coverage-guided fuzzing engines. AFL [135] uses a file to communicate the fuzzer generated input with the target program. The fuzzing device responds to read requests from device drivers with the contents of the file.

As mentioned in Section 3.1.1, when a USB device is attached to a computer, the USB driver framework reads the device descriptors and configuration descriptors and uses the appropriate driver to interact with it. However, depending on the implementation of the USB stack, the device descriptor and configuration descriptor may be read multiple times (e.g., the Linux kernel reads the device descriptor both before and after setting the address of the USB device). To improve fuzzing efficiency and considering that throughput is relatively low compared to simple user space fuzzing (see Section 3.5.3), these two requests are handled separately: they are loaded (either from a separate file or the fuzzer generated file) once when the fuzzing device is initialized and our framework responds with the same descriptors when requested. All other requests are served with bytes from the current position of the fuzzer generated file until no data is available, in which case, the device responds with no data. Note that as we are fuzzing the device drivers using data read from the device side, write operations to the device are ignored.

This design allows either *broad* fuzzing or *focused* fuzzing. By allowing the fuzzer to mutate the device and configuration descriptors (loading them from the fuzzer generated file), we can fuzz the common USB driver framework and drivers for a wide range of devices (broad fuzzing); by fixing the device and configuration descriptor to some specific device or class of devices (loading them from a separate configuration file), we can focus on fuzzing of a single driver (focused fuzzing). This flexibility enables different scenarios, e.g., it allows bug hunting in the USB driver framework and all deployed USB device drivers, or it can be used to test the driver of a specific USB device during the development phase. We demonstrate *focused fuzzing* on a USB webcam driver in Section 3.5.4.

3.3.2 Fuzzer – Guest System Communication

Like all existing fuzzers, the fuzzer component in USBFUZZ needs to communicate with the target code to exert control over tests, reap coverage information, and so forth. As shown in Figure 3.2, the fuzzer component runs outside the guest system and cannot gain information about the target system directly. The communication device is intended to facilitate the communication between the fuzzer and the guest system.

In a coverage-guided fuzzer, coverage information needs to be passed from the guest system to the fuzzer. To avoid repeated memory copy operations, we map the bitmap, which is a memory area in the fuzzer process, to the guest system using a QEMU communication device. After the guest system is fully initialized, the bitmap is mapped to the virtual memory space of the target kernel, to which the instrumented code in the target kernel can write the coverage information. As it is also a shared memory area in the fuzzer process, the coverage information is immediately accessible by the fuzzer component, avoiding memory copy operations.

In addition, the fuzzer component needs to synchronize with the user mode agent running in the guest system (see Section 3.3.3) in each fuzz test iteration. To avoid heavy-weight IPC operations, a control channel is added to the communication device to facilitate the synchronization between the user mode agent and the fuzzer component.

3.3.3 Test Execution and Monitoring

Existing kernel fuzzers execute tests using the process abstraction of the target kernel. They follow an iterative pattern where, for each test, a process is created, executed, monitored, and the fuzzer then waits for the termination of the process to detect the end of the test. In USBFUZZ, as tests are performed using the fuzzing device, in each iteration, a test starts with virtually attaching the (emulated) fuzzing device to the guest system. The kernel then receives a request for the new USB device that is handled by the low-end part of the kernel device management which loads the necessary drivers and initializes the device state. However, without support from the kernel through, e.g., process abstractions similar to the

`exit` system call, it is challenging to monitor the execution status (e.g., whether a kernel bug is triggered or not) of the kernel during its interaction with the device.

In USBFUZZ, we follow an empirical approach to monitor the execution of a test by the kernel: by checking the kernel's logged messages. For example, when a USB device is attached to the guest system, if the kernel is able to handle the inputs from the device, the kernel will log messages containing a set of keywords indicating the success or failure of the interaction with the device. Otherwise, if the kernel cannot handle the inputs from the device, the kernel will freeze or indicate that a bug was triggered. The USBFUZZ user mode agent component monitors the execution status of a test by scanning kernel logs from inside the virtualized target system, synchronizing its status with the fuzzer component so that it records bug triggering inputs and continues to the next iteration.

To avoid repeatedly booting the guest system for each iteration, USBFUZZ provides a persistent fuzzing technique, similar to other kernel fuzzers (syzkaller [47], TriforceAFL [50], trinity [56], or kAFL [101]), where a running target kernel is reused for multiple tests until it freezes, in which case, the fuzzer automatically restarts the kernel.

3.3.4 Coverage-Guided Fuzzing on Linux

So far, the USBFUZZ framework provides basic support for fuzzing USB device drivers on different OSes. However, to enable coverage-guided fuzzing, the system must collect execution coverage. A coverage-guided fuzzer keeps track of code coverage exercised by test inputs and mutates interesting inputs which trigger new code paths.

Coverage collection is challenging for driver code in kernel space. On one hand, inputs from the device side may trigger code executions in different contexts, because drivers may contain code running in interrupts and kernel threads. On the other hand, due to the kernel performing multitasking, code executed in a single thread may be preempted by other unrelated code execution triggered by timer interrupts or task scheduling. To the best of our knowledge, the Linux kernel only supports coverage collection by means of static instrumentation through `kcov` [125]. However, `kcov` coverage collection is limited to a single process, ignoring interrupt contexts and kernel threads. Extending the static instrumentation

of `kcov`, we devised an AFL-style edge coverage scheme to collect coverage in USB device drivers of the Linux kernel. To collect coverage across different contexts, (i) the previous executed block is saved in the context of each thread of code execution (interrupts or kernel threads), so that edge transitions are not mangled by preempted flows of code execution; and (ii) instrumentation is limited to related code: USB core, host controller drivers, and USB drivers.

3.4 Implementation Details

The implementation of the USBFUZZ framework extends several open source components including QEMU [11, 112] (where we implement the communication device and the emulated USB device), AFL [135] (which we modify to target USB devices by collecting coverage information from our virtualized kernel and interacting with our User Mode Agent), and `kcov` [125] (which we extend to track edge coverage across the full USB stack, including interrupt contexts). We implement the user mode agent from scratch. The workflow of the whole system, illustrating the interaction among the components, is presented in Figure 3.3. The implementation details of individual components are discussed in the following sections.

When the fuzzer starts, it allocates a memory area for the bitmap and exports it as a shared memory region, with which the communication device is initialized as QEMU starts. After the target kernel is booted, the user mode agent runs and notifies the fuzzer to start testing.

In each iteration of the fuzzing loop, the fuzzer starts a test by virtually attaching the fuzzing device to the target system. With the attachment of the fuzzing device, the kernel starts its interaction with the device and loads appropriate USB drivers for it. The loaded USB driver is tested with the fuzz input as it interacts with the fuzzing device. The user mode agent monitors execution by scanning the kernel log and notifies the fuzzer of the result of the test. The fuzzer completes the test by virtually detaching the fuzzing device from the target system.

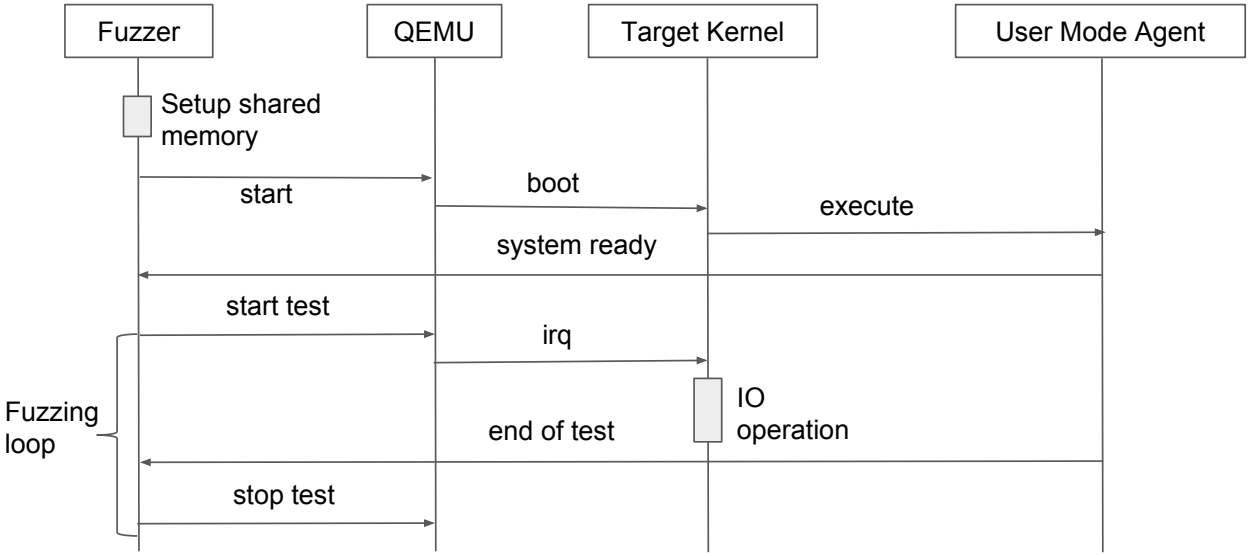


Figure 3.3. Workflow of USBFUZZ.

3.4.1 Communication Device

The communication device in USBFUZZ facilitates lightweight communication between the fuzzer component and the target system, which includes sharing the bitmap area and synchronization between the user mode agent and the fuzzer component. The implementation of the communication device is built on the IVSHMEM (Inter-VM shared memory) device [113], which is an emulated PCI device in QEMU. The shared memory region from the fuzzer component is exported to the guest system as a memory area in IVSHMEM device and mapped to the virtual memory space of the guest system. One register (BAR2, the Base Address Register for a memory or IO space) is used for the communication channel between the fuzzer component and the user mode agent.

3.4.2 Fuzzer

The fuzzer uses two pipes to communicate with the VM: a control pipe and a status pipe. The fuzzer starts a test by sending a message to the VM via the control pipe, and it receives execution status information from the VM via the status pipe.

On the VM side, two callbacks are registered for the purpose of interfacing with the fuzzer component. One callback attaches a new instance of the fuzzing device to the hypervisor with the fuzzer-generated input when a new message is received from the control pipe. When execution status information is received from the user mode agent via the communication device, the other callback detaches the fuzzing device from the hypervisor and forwards execution status information to the fuzzer via the status pipe.

3.4.3 Fuzzing Device

The fuzzing device is the key component in USBFUZZ that enables fuzzing of the hardware input space of the kernel. It is implemented as an emulated USB device in the QEMU device emulation framework and mimics an attacker-controlled malicious device in real-world scenarios.

Hypervisors intercept all device read/write requests from the guest kernel. Every read-/write operation from the kernel of the guest OS is dispatched to a registered function in the emulated device implementation, which performs actions and returns data to the kernel following the hardware specification.

The fuzzing device is implemented by registering "read" functions which forward the fuzzer-generated data to the kernel. To be more specific, the bytes read by device drivers are mapped sequentially to the fuzzer-generated input, except the device and configuration descriptors, which are handled separately (as mentioned in Section 3.3.1).

3.4.4 User Mode Agent

The user mode agent is designed to be run as a daemon process in the guest OS and is automatically started when the target OS boots up. It monitors the execution status of tests based on the kernel log and passes information to the fuzzer via the communication device. After initialization, it notifies the fuzzer that the target kernel is ready to be tested.

On Linux and FreeBSD, our user mode agent component monitors the kernel log file (`/dev/kmsg` in Linux, `/dev/klog` in FreeBSD), and scans it for error messages indicating a kernel bug or end of a test. If either event is detected, it notifies the fuzzer—using the device file exported to user space by the communication device driver—to stop the current iteration and proceed to the next one. The set of error messages is borrowed from the report package [88] of syzkaller. On Windows and MacOS, due to the lack of a clear signal from the kernel when devices are attached/detached, our user mode agent uses a fixed timeout (1 second on MacOS and 5 seconds on Windows) to let the device properly initialize.

3.4.5 Adapting Linux kcov

To apply coverage-guided fuzzing on USB drivers for the Linux kernel, we use static instrumentation to collect coverage from the target kernel. The implementation is adapted from `kcov` [125] which is already supported by the Linux kernel with the following modifications to accommodate our design.

```
index = (hash(IP) ^ hash(prev_loc))%BITMAP_SIZE;
bitmap[index] ++;
prev_loc = IP;
```

Listing 3.2. Instrumentation used in USBFUZZ

USBFUZZ implements an AFL-style [135] edge coverage scheme by extending `kcov`. Our modification supports multiple paths of execution across multiple threads and interrupt handlers, untangling non-determinism. We save the previous block whenever non-determinism happens. For processes, we save `prev_loc` (see Listing 3.2) in the `struct task` (the data structure for the process control block in the Linux kernel), and for interrupt handlers we save `prev_loc` on the stack. Whenever non-determinism happens, the current previous location is spilled (in the `struct task` for kernel threads, or on the stack for interrupt handlers) and set to a well-defined location in the coverage map, untangling non-determinism to specific locations. When execution resumes, the spilled `prev_loc` is restored. Note that this careful design allows us to keep track of the execution of interrupts (and nested interrupts) and separates their coverage without polluting the coverage map through false updates.

The instrumented code is modified to write the coverage information to the memory area of the communication device, instead of the per-process buffer. The Linux build system is modified to limit the instrumentation to only code of interest. In our evaluation, we restrict coverage tracking to anything related to the USB subsystem, including drivers for both host controllers and devices.

3.5 Evaluation

We evaluate various aspects of USBFUZZ. First, we perform an extensive evaluation of our coverage-guided fuzzing implementation on the USB framework and its device drivers (broad fuzzing) in the Linux kernel. Section 4.7.2 presents the discovered bugs, and Section 3.5.3 presents the performance analysis. Second, we compare USBFUZZ to the usb-fuzzer extension of syzkaller based on code coverage and bug discovery capabilities (Section 3.5.2). In Section 3.5.4, we demonstrate the flexibility of USBFUZZ by fuzzing (i) USB

drivers in FreeBSD, MacOS, and Windows (broad fuzzing); and (ii) a webcam driver (focused fuzzing). Finally, we showcase one of the discovered bugs in the USB core framework of the Linux kernel (Section 3.5.5).

Hardware and Software Environment. We execute our evaluation on a small cluster in which each of the four nodes runs Ubuntu 16.04 LTS with a KVM hypervisor. Each node is equipped with 32 GB of memory and an Intel i7-6700K processor with Intel VT [55] support.

Guest OS Preparation. To evaluate FreeBSD, Windows, and MacOS, we use VM images with unmodified kernels and a user mode agent component running in userspace. When evaluating Linux, the target kernel is built with the following customization: (i) we adapt `kcov` as mentioned in Section 3.4.5; (ii) we configure all USB drivers as built-in; (iii) we enable kernel address sanitizer (KASAN) [24, 25] to improve bug detection capability. At runtime, to detect abnormal behavior triggered by the tests, we configure the kernel to panic in case of "oops" or print warnings by customizing kernel parameters [121].

Seed Preparation. To start fuzzing, we create a set of USB device descriptors as seeds. We leverage the set of expected identifiers (of devices, vendors, products, and protocols) and matching rules of supported devices that syzkaller [47] extracted from the Linux kernel [116]. A script converts the data into a set of files containing device and configuration descriptors as fuzzing seeds.

3.5.1 Bug Finding

To show the ability of USBFUZZ to find bugs, we ran USBFUZZ on 9 recent versions of the Linux kernel: v4.14.81, v4.15, v4.16, v4.17, v4.18.19, v4.19, v4.19.1, v4.19.2, and v4.20-rc2 (the latest version at the time of evaluation). Each version was fuzzed with four instances for roughly four weeks (reaching, on average, approximately 2.8 million executions) using our small fuzzing cluster.

Table 3.1 summarizes all of the bugs USBFUZZ found in our evaluation. In total, 47 unique bugs were found. Of these 47 bugs, 36 are memory bugs detected by KASAN [25], including double-free (2), NULL pointer dereference (8), general protection error (6), out-

Table 3.1. Bug Classification

Type	Bug Symptom	#
Memory Bugs (36)	double-free	2
	NULL pointer dereference	8
	general protection	6
	slab-out-of-bounds access	6
	use-after-free access	14
Unexpected state reached (11)	WARNING	9
	BUG	2

of-bounds memory access (6), and use-after-free (14). 16 of these memory bugs are new and have never been reported. The remaining 20 memory bugs were reported before, and so we used them as part of our ground truth testing. Memory bugs detected by KASAN are serious and may potentially be used to launch attacks. For example, NULL pointer dereference bugs lead to a crash, resulting in denial of service. Other types of memory violations such as use-after-free, out-of-bounds read/write, and double frees can be used to compromise the system through a code execution attack or to leak information. We discuss one of our discovered memory bugs and analyze its security impact in detail in our case study in Section 3.5.5.

The remaining 11 bugs (WARNING, BUG) are caused by execution of (potentially) dangerous statements (e.g., assertion errors) in the kernel, which usually represent unexpected kernel states, a situation that developers may be aware of but that is not yet properly handled. The impact of such bugs is hard to evaluate in general without a case-by-case study. However, providing a witness of such bugs enables developers to reproduce these bugs and to assess their impact.

Bug Disclosure. We worked with the Linux and Android security teams on disclosing and fixing all discovered vulnerabilities, focusing first on the memory bugs. Table 3.2 shows the 11 **new** *memory bugs* that we fixed so far. These new bugs were dispersed in different USB subsystems (USB Core, USB Sound, or USB Network) or individual device drivers. From these 11 new bugs, we have received 10 CVEs. The remaining bugs fall into two classes: those still under embargo/being disclosed and those that were concurrently found and reported by other researchers. Note that our approach of also supplying patches for

Table 3.2. USBFUZZ's new memory bugs in 9 recent Linux kernels (SOOB: slab-out-of-bounds, UAF: use-after-free) that we fixed.

Kernel bug summary	Kernel Subsystem	Confirmed Version	Fixed	CVE
KASAN: SOOB Read in __usb_get_extra_descriptor	USB Core	4.14.81 - 4.20-rc2	✓	CVE-2018-20169
KASAN: UAF Write in usb_audio_probe	USB Sound	4.14.81 - 4.20-rc2	✓	CVE-2018-19824
KASAN: SOOB Read in build_audio_procunit	USB Sound	4.14.81 - 4.20-rc2	✓	CVE-2018-20344
KASAN: SOOB Read in parse_audio_input_terminal	USB Sound	4.14.81 - 4.18	✓	
KASAN: SOOB Read in parse_audio_mixer_unit	USB Sound	4.14.81 - 4.20-rc2	✓	CVE-2019-15117
KASAN: SOOB Read in create_composite_quirks	USB Sound	4.14.81 - 4.20-rc2	✓	
KASAN: SOOB Write in check_input_term	USB Sound	4.14.81 - 4.20-rc2	✓	CVE-2019-15118
KASAN: SOOB Read in hso_get_config_data	USB Network	4.14.81 - 4.20-rc2	✓	CVE-2018-19985
KASAN: NULL deref in ath6kl_usb_alloc_urb_from_pipe	Device Driver	4.14.81 - 4.20-rc2	✓	CVE-2019-15098
KASAN: NULL deref in ath10k_usb_alloc_urb_from_pipe	Device Driver	4.14.81 - 4.20-rc2	✓	CVE-2019-15099
KASAN: SOOB Read in lan78xx_probe	Device Driver	4.14.81 - 4.17	✓	
KASAN: double free in rsi_91x_deinit	Device Driver	4.17 - 4.20-rc2	✓	CVE-2019-15504
KASAN: OOB access bug in technisat_usb2_get_ir	Device Driver	4.14.81 - 4.20-rc2	✓	CVE-2019-15505

the discovered bugs reduces the burden on the kernel developers when fixing the reported vulnerabilities.

3.5.2 Comparison with Syzkaller

Due to challenges in porting the kernel-internal components of syzkaller, we had to use a version of the Linux kernel that is supported by syzkaller. We settled on version v5.5.0 [45], as it is maintained by the syzkaller developers. In this version, many of the reported USB vulnerabilities had already been fixed. Note that USBFUZZ does not require any kernel components and supports all recent Linux kernels, simplifying porting and maintenance. In this syzkaller comparison we evaluate coverage and bug finding effectiveness, running five 3-day campaigns of both USBFUZZ and syzkaller.

Bug Finding. In this heavily patched version of the Linux kernel, USBFUZZ found 1 bug in each run within the first day and syzkaller found 3 different bugs (2 runs found 2, 3

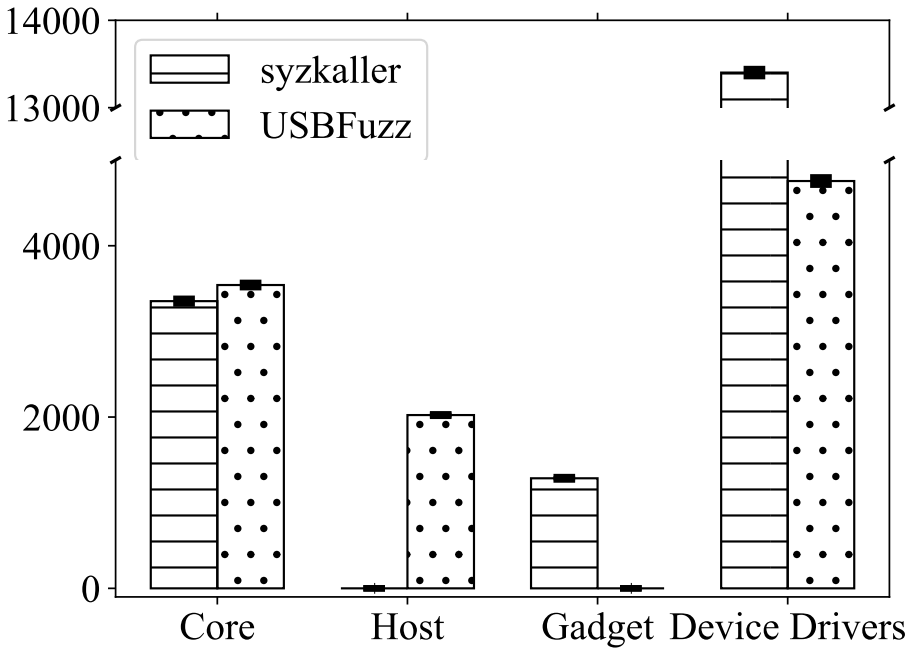


Figure 3.4. Comparison of line coverage between syzkaller and USBFUZZ in USB Core, host controller drivers, gadget subsystem, and other device drivers.

Table 3.3. Comparison of line, function, and branch coverage in the Linux kernel between syzkaller and USBFUZZ. The results are shown as the average of 5 runs.

	Line (%)	Function (%)	Branch (%)
syzkaller	18,039 (4.5)	1,324 (5.6)	7,259 (3.2)
USBFuzz	10,325 (2.5)	813 (3.5)	4,564 (2.0)

runs found 3). The bug USBFUZZ found is a new bug that triggers a *BUG_ON* statement in a USB camera driver [68]. The bugs found by syzkaller trigger WARNING statements in different USB drivers.

Code Coverage. We collected accumulated code coverage in the USB related code (including the USB core framework, host controller drivers, gadget subsystem, and other device drivers) by replaying inputs generated from both fuzzers. The line, function, and branch coverage of 5 runs are shown in Table 3.3. Overall, syzkaller outperforms USBFUZZ on maximizing code coverage. We attribute the better coverage to the manual analysis of the kernel code and custom tailoring the individual generated USB messages to the different USB drivers and protocols. The manual effort results in messages adhering more closely to the standard [117]—at a high engineering cost.

Table 3.3 shows that both syzkaller and USBFUZZ only triggered limited code coverage. There are three reasons: (i) some drivers are not tested at all; (ii) some code (function routines) can be triggered only by operations from userspace, and are thus not covered; (iii) some host controller drivers can only be covered with a specific emulated host controller.

Figure 3.4 demonstrates the differences between USBFUZZ and syzkaller. First, syzkaller triggered zero coverage in the host controller drivers. This is because syzkaller uses a USB gadget and a software host controller (dummy HCD) while USBFUZZ leverages an emulated USB device to feed fuzzer generated inputs to drivers. Though syzkaller may find bugs in the USB gadget subsystem, which is only used in embedded systems as firmware of USB devices and not deployed o1n PCs, it cannot find bugs in host controller drivers. We show a bug found in XHCI driver in our extended evaluation in Section 3.5.4.

Syzkaller achieves better overall coverage for device drivers due to the large amount of individual test cases that are fine-tuned. These syzkaller test cases can be reused for focused, per device fuzzing in USBFUZZ to extend coverage. USBFUZZ achieves better coverage in USB core, which contains common routines for handling data from the device side. This is caused by the difference in the input generation engines of the two fuzzers. As a generational fuzzer, syzkaller's input generation engine always generates valid values for some data fields, thus prohibiting it from finding bugs triggered by inputs that violate the expected values in these fields. USBFUZZ, on the other hand, generates inputs triggering such code paths. Note

that the driver in which USBFUZZ found a bug was previously tested by syzkaller. However, as the inputs it generated are well-formed, the bug was missed. We show an example of this in Section 3.5.5.

In summary, syzkaller leverages manual engineering to improve input generation for specific targets but misses bugs that are not standard compliant or outside of where the input is fed into the system. USBFUZZ follows an out-of-the box approach where data is fed into the unmodified subsystem, allowing it to trigger broader bugs. These two systems are therefore complementary and find different types of bugs and should be used concurrently. As future work, we want to test the combination of the input generation engines, sharing seeds between the two.

3.5.3 Performance Analysis

To assess the performance of USBFUZZ we evaluate execution speed and analyse time spent in different fuzzing phases.

Fuzzing Throughput. Figure 3.5 shows the execution speed of USBFUZZ in a sampled period of 50 hours while running on Linux 4.16. The figure demonstrates that USBFUZZ achieves a fuzzing throughput ranging from 0.1–2.6 exec/sec, much lower than that of userspace fuzzers where the same hardware setup achieves up to thousands of executions per second. Note the low fuzzing throughput in this scenario is mostly not caused by USBFUZZ, because tests on USB drivers run much longer than userspace programs. E.g., our experiment with physical USB devices shows that it takes more than 4 seconds to fully recognize a USB flash drive on a physical machine. A similar throughput (0.1–2.5 exec/sec) is observed in syzkaller and shown in Figure 3.6.

Overhead Breakdown. To quantify the time spent for each executed test, and to evaluate possible improvements in fuzzing throughput, we performed an in-depth investigation on the time spent at each stage of a test. As mentioned in Section 4.6, a test is divided into 3 stages, (i) virtually attaching the fuzzing device to the VM; (ii) test execution; and (iii) detaching the fuzzing device. We measure the time used for attaching/detaching, and the time used in running a test when device drivers perform IO operations. The result is shown

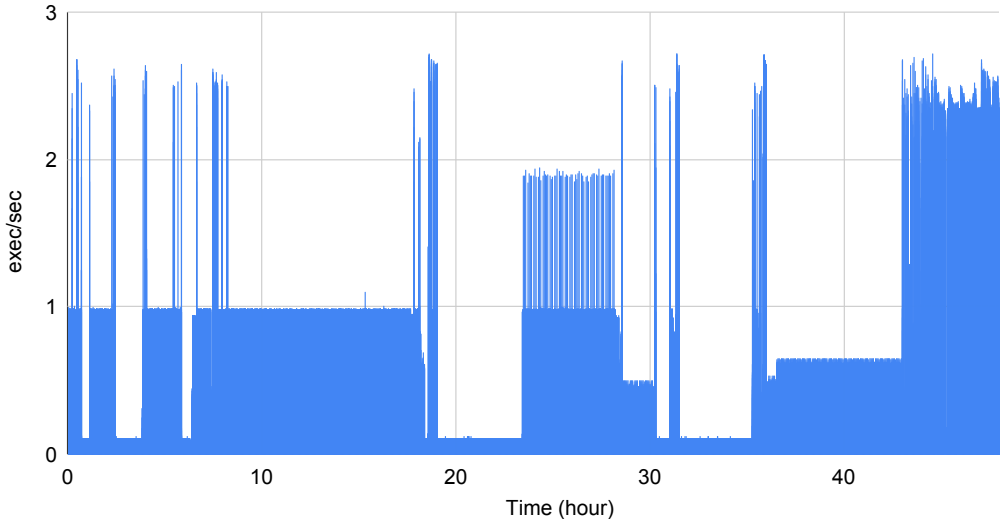


Figure 3.5. A sample of execution speed of USBFUZZ

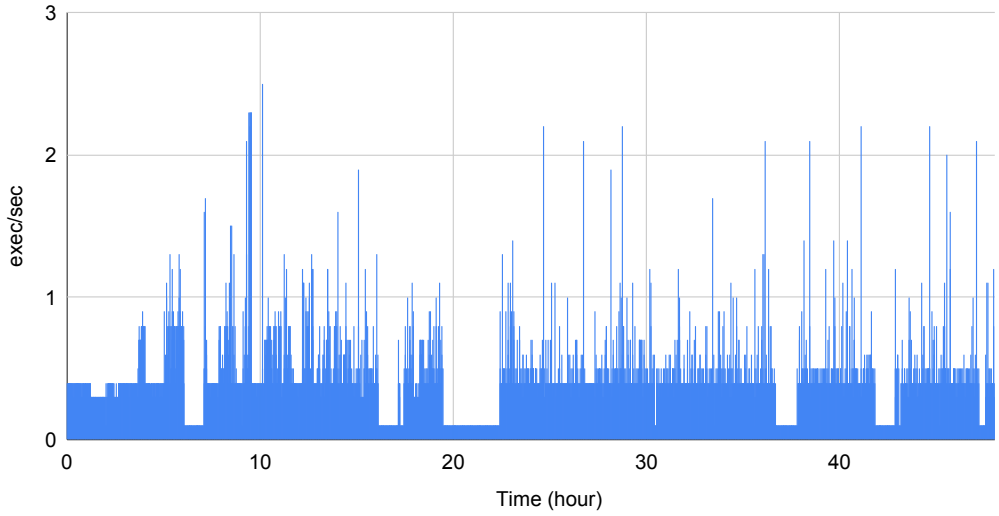


Figure 3.6. A sample of execution speed of syzkaller

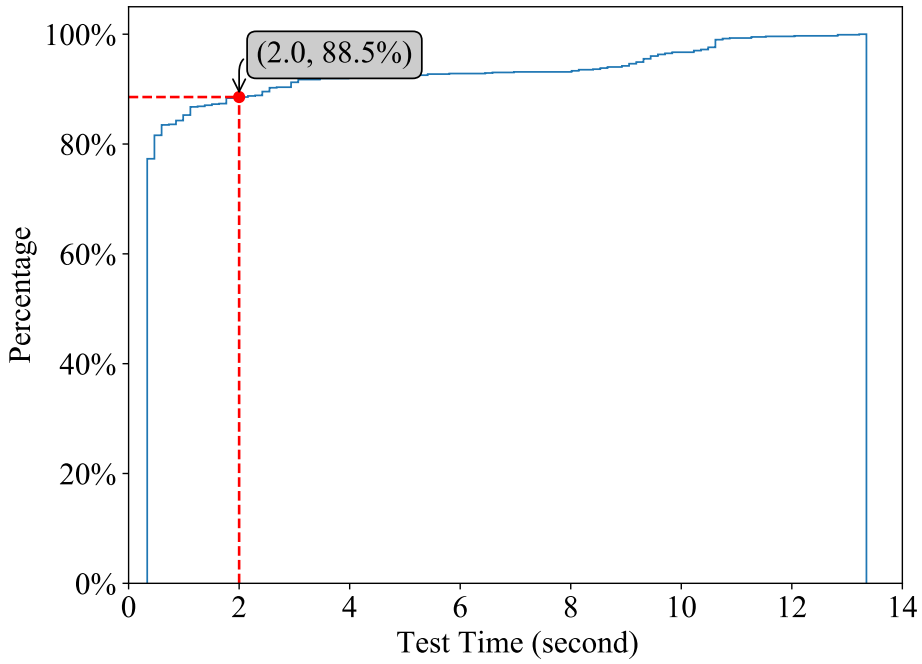


Figure 3.7. Cumulative distribution of test run time, collected by tracing the inputs generated by USBFUZZ.

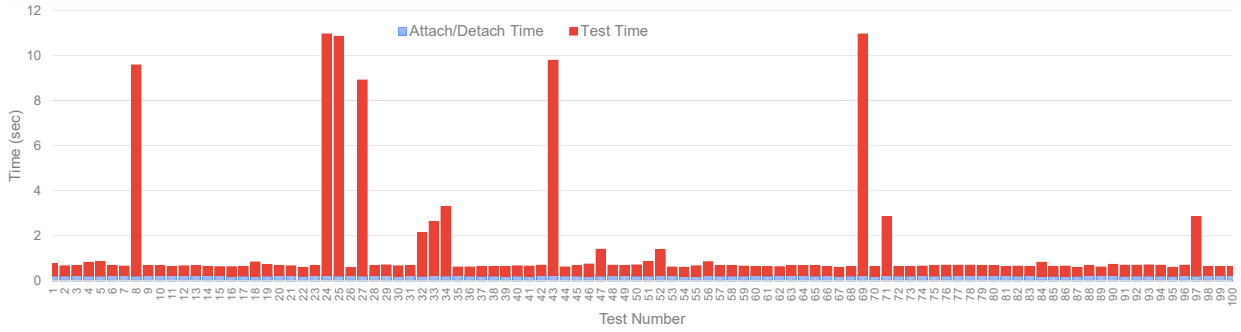


Figure 3.8. Execution Time Breakdown of 100 randomly chosen tests. The axes denote test number and execution time. Blue and red bars represent time used in attaching/detaching the emulated device to the VM and the time spent in testing respectively.

in Figure 3.8. The blue line and red line show the time used in the attach/detach operations (added together) and the time used in tests respectively. From Figure 3.8, the time used in these attach/detach operations remains stable at about 0.22 second, while the time used by tests varies from test to test, ranging from 0.2 to more than 10 seconds.

Manual investigation on the test cases shows that the time a test takes depends on the quality of input. If the input fails the first check on the sanity of the device descriptor, it finishes very quickly. If the emulated device passes initial sanity checks and is bound to a driver, the execution time of a test depends on the driver implementation. Typically longer tests trigger more complex code paths in device drivers. Figure 3.7 depicts the runtime distribution of tests generated by USBFUZZ. It shows that about 11% of the generated tests last longer than 2 seconds.

We also evaluated the overhead caused by the user mode agent component. We measured the time used to run tests on a base system with the user mode agent running and that without user mode agent, a comparison shows that the difference is roughly 0.01 second, which is negligible compared to the overall test execution time.

Though the overhead of attach/detach operations is negligible for long tests, it accounts for about 50% of the total execution time of short tests. As the emulated device is allocated/deallocated before/after the test in each iteration, this overhead can be reduced by caching the emulated device and performing only necessary initialization operations. We leave this optimization as future work.

3.5.4 USBFUZZ Flexibility

To demonstrate the benefit of portability and flexibility of the USBFUZZ framework, we performed two extended evaluations: (i) fuzzing FreeBSD, MacOS, and Windows; (ii) focused fuzzing a USB webcam driver.

Fuzzing FreeBSD, MacOS, and Windows. Leveraging the portability of a device emulation-based solution to feed fuzzer-generated inputs to device drivers, we extended our evaluation to FreeBSD 12 (the latest release), MacOS 10.15 Catalina (the latest release) and Windows (both version 8 and 10, with most recent security updates installed). After porting the user mode agent and the device driver of the communication device we apply dumb fuzzing on these OSes.

Fuzzing drivers on these OSes is more challenging than the Linux kernel due to the lack of support infrastructure. These OSes support neither KASAN, other sanitizers, nor coverage-

based collection of executions. The lack of a memory-based sanitizer means our fuzzer only discovers bugs that trigger exceptions, and misses all bugs that silently corrupt memory. Because we cannot collect coverage information, our fuzzer cannot detect seeds that trigger new inputs.

To alleviate the second concern, the lack of coverage-guided optimization, we experiment with cross-pollination. To seed our dumb fuzzer, we reuse the inputs generated during our Linux kernel fuzzing campaign.

USBFUZZ found three bugs (two resulting unplanned restart and one resulting system freeze) on MacOS, and four bugs on Windows (resulting in a Blue Screen of Death, confirmed on both Window 8 and Windows 10) during the first day of evaluation. Additionally, one bug was found in a USB Bluetooth dongle driver on FreeBSD in two weeks. In this bug, the driver is trying to add an object to a finalized container.

Focused fuzzing on the LifeCam VX-800 driver. So far, we let the fuzzer create emulated USB peripherals as part of the input generation process. Here we want to show the capability of USBFUZZ of fuzzing focusing on a specific device. We extract the device and configuration descriptor from a real LifeCam VX-800 [70] webcam (with the lsusb [23] utility) and let USBFUZZ create a fake USB device based on that information, enabling the Linux kernel to detect and bind a video driver to it.

We extended the user mode agent to receive a picture from the webcam with streamer [122][2] using the emulated device. After fuzzing this targeted device for a few days with randomly generated inputs, we found another bug in the XHCI [129] driver of the Linux kernel. The buggy input triggers an infinite loop in the driver, in which the driver code keeps allocating memory in each iteration until the system runs out of memory.

USBFUZZ Flexibility. The bugs found in the FreeBSD, MacOS and Windows, and XHCI driver demonstrate the advantage of USBFUZZ compared to syzkaller's usb-fuzzer. As the implementation of usb-fuzzer only depends on the Linux kernel, it cannot be ported other OSes without a full reimplementation. Moreover, as usb-fuzzer injects fuzzer-generated inputs via a software host controller (dummy HCD [107]), it is unable to trigger bugs in drivers of physical host controllers.

[2]↑We execute the `streamer -f jpeg -o output.jpeg` command.

3.5.5 Case Study

In this section, we discuss a new bug USBFUZZ discovered in the USB core framework of the Linux kernel. In the USB standard, to enable extensions, a device is allowed to define other customized descriptors in addition to the standard descriptors. As the length of each descriptor varies, the USB standard defines the first two bytes of a descriptor to represent the length and type of a descriptor (as shown by `usb_descriptor_header` in Listing 3.3). All descriptors must follow the same format. For example, an OTG (USB On-The-Go, a recent extension which allows a USB device to act as a host [131]) descriptor (shown as `usb_otg_desciptor` in Listing 3.3) has three bytes and thus a correct OTG descriptor must start with a `0x03` byte.

Descriptors are read from the device, and therefore, cannot be trusted and must be sanitized. In the Linux kernel, `__usb_get_extra_descriptor` is one of the functions used by the USB core driver to parse the customized descriptors. Listing 3.3 shows that the code simply scans the data (`buffer` argument) read from the device side. To match descriptors for a given type (`type` argument) it returns the first match.

When handling maliciously crafted descriptors, this implementation is vulnerable. By providing a descriptor that is shorter than its actual length, the attacker can trigger an out-of-bounds memory access. E.g., a two byte (invalid) OTG descriptor with the third byte missing will be accepted by `__usb_get_extra_descriptor` and treated as valid. If the missing field is accessed (e.g., the read of `bmAttributes` at line 30), an out-of-bounds memory access occurs.

Depending on how the missing fields are accessed, this vulnerability may be exploited in different ways. For example, reading the missing fields may allow information leakage. Similarly, writing to the missing fields corrupts memory, enabling more involved exploits (e.g., denial-of-service or code execution). Although our fuzzer only triggered an out-of-bounds read, an out-of-bounds write may also be possible.

```
1  struct usb_descriptor_header {
2    __u8  bLength;
3    __u8  bDescriptorType;
4  } __attribute__ ((packed));
5  struct usb_otg_descriptor {
6    __u8  bLength;
7    __u8  bDescriptorType;
8    __u8  bmAttributes;
9  } __attribute__ ((packed));
10 int __usb_get_extra_descriptor(char *buffer, unsigned size, char type,
       void **ptr) {
11   struct usb_descriptor_header *header;
12   while (size >= sizeof(struct usb_descriptor_header)) {
13     header = (struct usb_descriptor_header *)buffer;
14     if (header->bLength < 2) {
15       printk("%s: bogus descriptor...\n", ...)
16     }
17     if (header->bDescriptorType == type) {
18       *ptr = header;
19       return 0;
20     }
21     buffer += header->bLength;
22     size -= header->bLength;
23   }
24   return -1;
25 }
26 static int usb_enumerate_device_otg(struct usb_device *udev) {
27   // ......
28   struct usb_otg_descriptor *desc = NULL;
29   err=__usb_get_extra_descriptor(udev->rawdescriptors[0], le16_to_cpu(udev
       ->config[0].desc.wTotalLength), USB_DT_OTG, (void **) &desc);
30   if (err||!(desc->bmAttributes & USB_OTG_HNP))
31     return 0;
32   // ......
33 }
```

Listing 3.3. Out-of-bounds vulnerability in the Linux USB core framework. The two byte descriptor (`0x02`, `USB_DT_OTG`) is accepted by `__usb_get_extra_descriptor` as three byte `usb_otg_descriptor`. Triggering an out-of-bounds access when the missing field `bmAttributes` is accessed at line 30.

3.5.6 Fuzzing Other Peripheral Interfaces

Peripheral interfaces represent a challenging attack surface. USBFUZZ is extensible to other peripheral interfaces supported by QEMU. To add support for a new peripheral interface in USBFUZZ, an analyst needs to: (i) implement a fuzzing device for the interface and adapt its reading operations to forward fuzzer generated data to the driver under test; (ii) adapt the fuzzer to start/stop a test by attaching/detaching the new fuzzing device to the VM; and (iii) adapt the user mode agent component to detect the end of tests based on the kernel log.

The SD card [5] is an interface that is well supported by QEMU and exposes a similar security threat as USB. SD cards are common on many commodity PCs and embedded devices. We extended USBFUZZ to implement SD card driver fuzzing. The implementation required few code changes: 1,000 LoC to implement the fuzzing device, 10 LoC to adapt the fuzzer, and 20 LoC to adapt the user-mode agent.

After adapting, we fuzzed the SD card interface for 72 hours. As the SD protocol is much simpler than USB (with fixed commands and lengths), and there are only a limited number of drivers, we did not discover any bugs after running several fuzzing campaigns on Linux and Windows.

3.6 Related Work

In this section, we discuss related work that aims at securing/protecting host OS from malicious devices.

Defense Mechanisms. As an alternative to securing kernel by finding and fixing bugs, defense mechanisms stop active exploitation. For example, Cinch [3] protects the kernel by running the device drivers in an isolated virtualization environment, sandboxing potentially buggy kernel drivers and sanitizing the interaction between kernel and driver. SUD [12] protects the kernel from vulnerable device drivers by isolating the driver code in userspace processes and confining its interactions with the device using IOMMU. Rule-based authorization policies (e.g., USBGuard [91]) or USB Firewalls (e.g., LBM [118] and USBFILTER [119]) work by blocking known malicious data packets from the device side.

Table 3.4. A comparison of USBFUZZ with related tools. The "Cov" column shows support for coverage-guided fuzzing. The "Data Inj" column indicates how data is injected to drivers: through the device interface (Device) or a modified API at a certain software layer (API). The "HD Dep" and "Portability" columns denote hardware dependency and portability across different platforms.

Tools	Cov	Data Inj	HD Dep	Portability
TTWE	✗	Device	✓	✓
vUSBf	✗	Device	✗	✓
umap2	✗	Device	✓	✓
usb-fuzzer	✓	API	✗	✗
USBFuzz	✓	Device	✗	✓

Cinch [3] and SUD [12] rely heavily on hardware support (e.g., virtualization and IOMMU modules). Though their effectiveness has been demonstrated, they are not used due to their inherent limitations and complexities. Rule-based authorization policies or USB Firewalls may either restrict access to only known devices, or drop known malicious packets, thus they can defend against known attacks but potentially miss unknown attacks. These mitigations protect the target system against exploitation but do not address the underlying vulnerabilities. USBFUZZ secures the target systems by discovering vulnerabilities, allowing developers to fix them.

Testing Device Drivers. We categorize existing device driver fuzzing work along several dimensions: support for coverage-guided fuzzing, how to inject fuzzed device data into tested drivers, and hardware dependency and portability across platforms. Support of coverage-guided fuzzing influences the effectiveness of bug finding, and the approach to inject device data into target drivers determines the portability. Hardware dependency incurs additional hardware costs.

Table 3.4 summarizes related work. Tools such as TTWE [124] and umap2 [51] depend on physical devices and do not support coverage-guided fuzzing. While eliminating hardware dependency through an emulated device interface for data injection, vUSBf [100] does not support coverage-guided fuzzing. usb-fuzzer [42] (a syzkaller [47] extension) supports coverage-guided fuzzing, and passes the fuzzer generated inputs to device drivers through extended system calls. However, its implementation depends on modifications to modules

(the gadgetfs [81] and dummy-hcd [107] modules) in the USB stack of the Linux kernel, and is thus not portable. In contrast, USBFUZZ is portable across different platforms and integrates coverage feedback (whenever the kernel exports it).

Sylvester Keil et al. proposed a fuzzer for WiFi drivers based on an emulated device [59]. While they also emulate a device, their system does not support coverage-guided fuzzing. They focus on emulating the functions of a single WiFi chip (the Atheros AR5212 [9]). As the hardware and firmware are closed source, they reverse engineered the necessary components. USBFUZZ, in comparison, does not require reverse engineering of firmware and supports all USB drivers in the kernel. In concurrent work, PeriScope [105] proposes to apply coverage-guided fuzzing on WiFi drivers by modifying DMA and MMIO APIs in the kernel. IoTFuzzer [19] targets memory vulnerabilities in the firmware of IoT devices. These tools either have additional dependencies on physical devices, or cannot leverage coverage feedback to guide their fuzzing. Additionally, the AVATAR [133] platform enables dynamic analysis of drivers by orchestrating the execution of an emulator with the real hardware.

Symbolic Execution. The S2E [20] platform adds selective symbolic execution support to QEMU. Several tools extend S2E to analyze device drivers by converting the data read from the device side into symbolic values (e.g, SymDrive [97] and DDT [63]). Potus [80] similarly uses symbolic execution to inject faulty data into USB device drivers.

Like our approach, symbolic execution eliminates hardware dependencies. However, it is limited by high overhead and scalability due to path explosion and constraint solving cost. Further, Potus is controlled by operations from userspace, thus *probe* routines are out of scope. In contrast, USBFUZZ follows a dynamic approach, avoiding these limitations and targets both *probe* routines and function routines.

3.7 Summary

The USB interface represents an attack surface, through which software vulnerabilities in the host OS can be exploited. Existing USB fuzzers are inefficient (e.g., dumb fuzzers like vUSBf), not portable (e.g., syzkaller usb-fuzzer), and only reach probe functions of drivers. We propose USBFUZZ, a flexible and modular framework to fuzz USB drivers in

OS kernels. USBFUZZ is portable to fuzz USB drivers on different OSes, leveraging coverage-guided fuzzing on Linux and dumb fuzzing on other kernels where coverage collection is not yet supported. USBFUZZ enables broad fuzzing (targeting the full USB subsystem and a wide range of USB drivers) and focused fuzzing on a specific device's driver.

Based on the USBFUZZ framework, we applied coverage-guided fuzzing (the state-of-art fuzzing technique) on the Linux kernel USB stack and drivers. In a preliminary evaluation on nine recent versions of the Linux kernel, we found 16 new memory bugs in kernels which have been extensively fuzzed. Reusing the generated seeds from the Linux campaign, we leverage USBFUZZ for dumb fuzzing on USB drivers in the FreeBSD, MacOS and Windows. To date we have found one bug in FreeBSD, three bugs on MacOS and four bugs on Windows. Last, focusing on a USB webcam driver, we performed focused fuzzing and found another bug in the XHCI driver of the Linux kernel. So far we have fixed 11 new bugs and received 10 CVEs. USBFUZZ is available at https://github.com/HexHive/USBFuzz.

3.8 Discussion and Future Work

We discuss challenges that limit the effectiveness and efficiency of USBFUZZ. These problems are common in fuzzing kernels from the perspective of devices.

Device modeling. The behaviour of a device can be very complex. Depending on the capability of the device, different behaviours can trigger different code paths in the kernel side. For example, a device can raise an interrupt signal at an unexpected time, or it can perform malicious read/write from/to the memory space of the host system. Even for USB devices, where the case is relatively simple, and the device is only allowed to send data to the host side when requested, the device can respond with more or less data than expected, or respond with data for a specific request while ignoring other requests. In the current implementation of USBFUZZ, the fuzzing device models the USB device in a simple way, and always responds with the specified size of data (at the protocol level) until all the fuzzer generated input is consumed. Thus it may miss bugs that can only be triggered with more complex device modeling. This is an open research problem.

Deep fuzzing. A device driver typically collaborates with other kernel subsystems and provides an abstract interface to other parts of the host OS. For example, an USB web camera driver registers the device to the video subsystem and exports a device file to user-space so that applications can use the device to record video streams or pictures. In such drivers, the driver code is passive, and its interaction with the device side is driven by requests from a user-space application. To achieve deep fuzzing of the driver code, a fuzzer needs to collaborate with user-space applications.

In the current stage, we focus on fuzzing the inputs read by drivers in the initialization code, which will be automatically executed when the device is attached. Additionally, our framework has initial support for user-space driven fuzzing. The user mode agent component may run customized code (following the libFuzzer [111] approach of creating customized fuzzers) after a USB device is recognized by the host OS, and the fuzzing device can be configured to emulate some specific device, see Section 3.5.4. As user-space applications vary from device to device and it requires device specific knowledge to write such applications, USBFUZZ (or any other purely coverage-guided fuzzer) cannot yet perform comprehensive deep fuzzing *automatically*.

Augmenting Input Generation. Though we use mutational fuzzing to generate inputs, and apply coverage-guided fuzzing on the Linux kernel, our fuzzing framework can also benefit from other input generation techniques. Like syzkaller [42], we can leverage format information to generate inputs that achieve better coverage. Or like PeriScope [105] we can trace the data communication between the drivers and devices and convert such traces to seeds to save the fuzzing cycles.

4. WEBGLFUZZER: FUZZING WEBGL INTERFACE VIA LOG GUIDED FUZZING

4.1 Introduction

The increasing demand for high performance 2D/3D graphics in web applications resulted in the incarnation of the WebGL (Web Graphics Library) [49] interface. WebGL is a standardized set of Javascript APIs supported in compatible browsers. Nowadays, it is widely supported by most desktop and mobile browsers. WebGL is also supported in mobile and desktop application frameworks (e.g., Android WebView [22] component, iOS WKWebView [26], or Electron [30]). This broad proliferation of easily accessible WebGL frameworks make them a prime target for attackers.

The introduction of WebGL brings a lot of security concerns as it exposes the underlying native graphic stacks to untrusted (malicious) remote users. Native graphic stacks, previously designed only for locally use, are now exposed to remote attackers. Native graphics stacks consist of highly complex components, e.g., GL libraries (OpenGL on Unix systems, Direct 3D on Windows etc) and GPU drivers. A single vulnerability in one of the components of the graphic stack may put millions of users at risk. The situation is aggravated by the fact that the graphic stacks are typically provided as close-source binary blobs by OEMs or independent third-party GPU vendors, and thus their security cannot be easily assessed. Many recent high severity CVEs in Chrome and Firefox demonstrate this unfortunate situation (e.g., CVE-2020-6492 [73] or CVE-2020-15675 [72]).

To proactively address the security risks of WebGL, we propose a fuzzing approach to uncover remotely triggerable bugs. As an API, the input space of the WebGL is huge lying along two dimensions: (1) ordering of API calls and (2) arguments passed to each API call. Typically, the execution of one API is dependent on both its arguments and the current internal states, which sets up the states required by another API. Thus, the input in both dimensions influences code coverage, and thus the effectiveness of a fuzzer. Given such a huge input space with interleaved dependencies, dumb fuzzing can only explore shallow code paths. To effectively reduce the search space, state-of-art fuzzing tools use code coverage as feedback to guide their input mutation [32, 47]. There are two inherent limitations with

coverage-guided fuzzing. First, it depends on precise code coverage information, which is challenging when analyzing the WebGL stack, because the software stack consists of several layers of libraries, different processes, and code running both in user and kernel space. In addition, several of the code blocks are closed source and proprietary. Second, coverage feedback does not indicate how to effectively extend coverage when performing mutation on the current input due to a-priori unknown complex inter-dependencies of API calls.

Considering these mentioned limitations of coverage guided fuzzing, We propose a alternative novel technique, called log message guided fuzzing, which drops dependency on code coverage and is able to perform more meaningful mutations. The idea of log message guided fuzzing is based on the key observation that during the execution of a WebGL program, when errors are detected, browsers emit meaningful log messages to aid developers in correcting the program. By intuition we expect the fuzzer to be capable of mutating the input following the feedback from such log messages like a human analyst.

To build a log message guided fuzzer, the key challenge is to build mutating rules for log messages, i.e., given a log message emitted by a browser, how to automatically figure out how to mutate the input. To this end, we studied the messages and learned that there are two types of log messages: (1) log messages indicating the invalidity of some arguments passed to the current API; (2) log messages indicating the invalidity of the internal WebGL program state. For log messages of type (1), the problem with the input lies in some arguments (called target arguments) passed to the API emitting the message. To identify the target arguments of this type of log messages, we use backward static taint analysis to track the dependencies of the checks the log message is conditioned on and the API arguments. Additionally, we build a set of mutating rules from the the semantic meaning of the log message using natural language analysis (e.g., if the log message complains the argument to be too large, the fuzzer chooses a smaller value instead of a random value). For log messages of type (2), the issue with the input lies in the internal WebGL program state, which can only be fixed with calls to some other APIs the current API is dependent on (called dependent API set). Thus for this type of messages, we compute its a dependent API set by statically analyzing which APIs update the internal states the checks of the log message is control dependent on. Using these information, instead of performing random mutation, the fuzzer focuses on mutating the

identified target argument using mutating rules built on semantic analysis on log message, or tries to fix the internal program state using APIs from its dependent set.

We implemented our log guided fuzzing technique in a prototype called WEBGLFUZZER. We are currently evaluating it with popular browsers (Chrome, Firefox and Safari) on both desktop (Linux, Windows and MacOS) and mobile (Andrdoid and iOS) OSes. So far we have found 6 bugs: 3 in Chrome, 2 in Safari and 1in Firefox. Out of these bugs, one is able to freeze X-Server on Linux.

In summary, the main contributions of this paper are as follows:

1. We propose a new log guided fuzzing technique, that is independent of code coverage collection and performs meaningful input mutations.

2. We develop a fuzzer for WebGL, WEBGLFUZZER, that implements our proposed log guided fuzzing technique.

3. We have found 6 bugs so far (in widely used browsers like Chrome, Firefox and Safari), all of which have been confirmed by developers and one freezes X-Server on Linux.

4.2 Background

4.2.1 WebGL Interface

Increasing demand for high performance GPU accelerated graphics rendering in web applications has led to the incarnation of standardized Web Graphis Library (WebGL) interface [49]. In a nutshell, WebGL is a set of Javascript APIs for 2D and 3D rendering and bound to the HTML5 Canvas element, to which the rendered result is saved. It is designed to closely conform to OpenGL ES API (a subset of OpenGL API tailored to embedded systems) [48], with similar constructs and using the same shading language (GLSL). So far there are two versions of WebGL specifications and each version exposes a large number of APIs (as shown in Table 4.1).

In implementation, WebGL is built on the native graphics stack to take advantage of GPU acceleration provided by the user's device. The Native graphic stacks consist of the GL libraries (OpenGL on Unix systems or Direct 3D on Windows) and GPU drivers, which are

Table 4.1. WebGL Specification Information

Version	Based On	# of APIs
WebGL 1	OpenGL ES 2	163
WebGL 2	OpenGL ES 3	333

typically provided by GPU vendors in proprietary binary blobs. Browsers detect the graphic stacks and associated GPUs and use available ones on the computer. E.g., The Chrome browser uses the ANGLE [83] component to transparently translate the WebGL calls to the hardware supported APIs available on the underlying OS (e.g., OpenGL/OpenGL ES on Unix based systems, Direct 3D on Windows), which in turn leverages the underlying GPU drivers for GPU accelerated graphics.

Nowadays, with the wide adoption of the web technology, support of WebGL has gone far beyond traditional web tools like browsers (desktop and mobile). E.g., application frameworks such as Election [30] and CORDOVA [89], are built based on the browser stack; the GUI of ChromeOS[92] is even completely built on the chrome browser. In addition, mobile app frameworks of Android and iOS also contain web gadgets (Android Webview [22] and iOS WKWebView [26]) to allow developers to load and execute web programs in their apps directly. Android WebView and iOS WKWebView are widely used to load Ads in mobile apps. WebGL is enabled by default in these frameworks and gadgets.

4.2.2 WebGL Security Concerns

Historically, applications (3D games etc) using graphic stacks pose limited security risks, as they run locally, and developers are well-known software vendors, thus can be trusted. However, as WebGL exposes the underlying graphics stack to untrusted remote users, its security implication is a great concern for browser vendors. Some vendors (e.g., Microsoft) were even very reluctant in supporting WebGL in the beginning [17]. The security concern of WebGL is rooted in the large, diverse, highly complex and potentially vulnerable graphic stacks provided by OEM or third party GPU vendors, which were not designed only for local use and not to fully defend against malicious users.

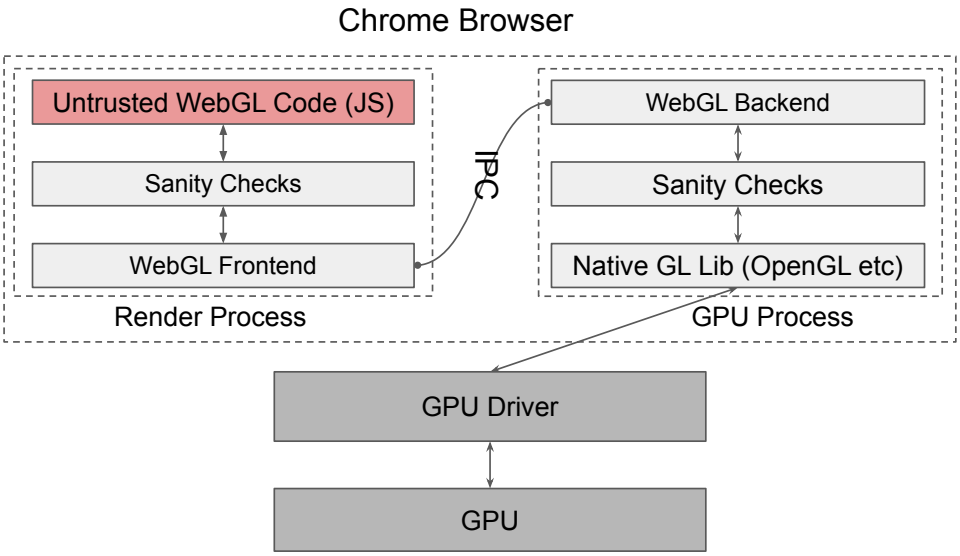


Figure 4.1. WebGL Implementation in Chrome browser

To mitigate the security concern, multiple runtime security defense mechanisms are deployed in browsers. E.g., in Chrome browser, (1) CFI is deployed to protect C++ virtual calls and C indirect calls [93]; (2) a multi-process design [8] is employed to compartmentalize the graphics stack code in a service process (GPU process) to prevent it from being accessed directly by untrusted WebGL code run in the render process (see Figure 4.1), invocation to the underlying graphic stack from WebGL is delegated to a service GPU process through IPC and shared memory; (3) a diverse set of runtime checks is used to vet the arguments passed to WebGL API calls and internal program state transition. Most of the runtime checks are derived from OpenGL ES specification [48], based on which WebGL is built. Other checks include ones added to prevent potential exploits, when new vulnerabilities are discovered. A comprehensive study on the types of deployed checks in Chrome is presented in Milkomeda [132].

In Chrome browser, runtime checks are deployed in both the render process and GPU process. As the checks for defending against known vulnerabilities may be driver specific and thus need to access the graphics stack, they are deployed only in the GPU process. The checks derived from OpenGL ES specification are deployed in both the render process and GPU process. One might wonder the rationale behind the implementation of the redundant

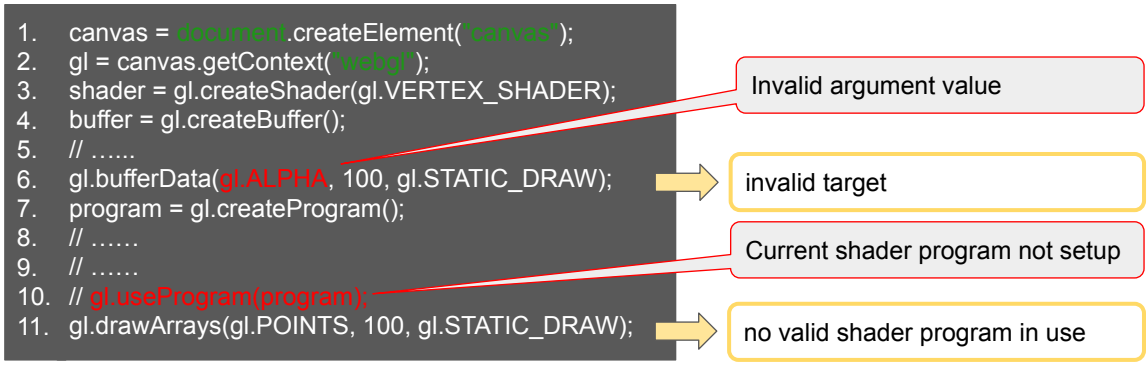


Figure 4.2. An example of runtime log messages emitted Chrome browser

checks. The deployment of checks in the render process avoid the performance penalties introduced by IPC with the GPU process; while the deployment of checks in the GPU process is based on the consideration that the integrity of the render process can not be guaranteed in realworld, as its execution may be under the control of an adversary leveraging vulnerabilities in it.

In addition to the security purpose, these checks also serve to provide developers with runtime feedbacks to aid WebGL program debugging. More specifically, when the checking code detects that some argument passed to the WebGL API is invalid, or the WebGL program in an invalid state to execute the current API, to console of the browser it will log some messages. There are 2 types of log messages: log message of type 1 indicating the invalidity of some arguement passed to the current API and type 2 indicating the invalidity of the internal program state. E.g., as shown in Figure 4.2, in line 6, if the first argument passed to `bufferData` is neither `GL_ELEMENT_ARRAY_BUFFER` nor `GL_ARRAY_BUFFER`, an "invalid target" message will be logged; in line 11 even if all the arguments passed to `drawArrays` are all correct, because the internal shader program is not properly setup when `drawArrays` is called, a "no valid shader program in use" message will be logged. Such log messages provide the developer with invaluable information to identify the problem and correct the program.

Even with these mitigation techniques, the security risks posed by WebGL can not be complete eliminated. Existing CFI systems demonstrate poor security because of the large permissible target sets over-approximated using practical static analysis techniques [40, 60]. Although compartmentalization avoids direct access to the underlying graphic stack, indirect

access is still unsafe, as demontrated by recently disclosed CVEs in Chrome [33, 75]. The deployed checks only serve as a passive approach to defend against known vulnerabilities, not even targeting unknown ones.

The risks exposed by WebGL and severity of potential consequences of attacks motivate the needs to uncover the anomalies in the underlying graphics stack that can be triggered from WebGL APIs. As the graphic stacks are provided by OEM or independent GPU vendors, most of which are provided only in binary blob, it is challenging to apply static analysis based approaches In this work, we propose a fuzz testing based approach to evaluate the security of the WebGL interface.

4.3 Threat Model

Our threat model consists of an adversary trying to gain control of the victim machine by executing a malicious WebGL program leveraging a vulnerability in the WebGL stack. To run the malicious WebGL program, the attacker can either lure the victim users to open a prepared web page containing the malicious WebGL program using phishing emails etc, or by shipping it with an Ad and execute it in the victim browsers through the Ads network.

4.4 Overview

4.4.1 Motivation

As a program interface, WebGL is similar to system calls, for which there are already some existing fuzzing tools [32, 47, 54, 56, 79]. Out of these tools, syzkaller [47] is the state-of-art, which internally uses code coverage as feedback to guide its input generation based on a set of manually investigated templates of argument format.

There are two limitations with the approach used by syzkaller. First, as a coverage guided fuzzing technique, it depends on precise code coverage information, which is challenging to be collected from for proprietary browsers (e.g., Safari) and graphic stack (dynamic instrumentation could be used, but would incur prohibitively high runtime overhead). Even in opensource browser and graphic stack, precise coverage collection is still challenging due to multi-process security design. E.g., in the Chrome browser, the execution of WebGL

APIs spans in the render process and GPU process. The render process and GPU process communicate with each other through asynchronous IPC, thus collecting precise code coverage triggered by WebGL APIs is not possible without nontrivial engineering effort (e.g., by modifying the source code in the target). Second, coverage feedback is not indicative of where and how to mutate the input to effectively expand coverage. Syzkaller only uses code coverage as a metric to evaluate the quality of inputs chooses an input to mutate based on the quality of input in each iteration. When performing mutation on the selected input, as code coverage does not contain any information such as which part of the input is invalid and rejected by the target code, Syzkaller simply performs blind and random mutation on the selected input.

Log Guided Mutation. Our key observation is that browsers log meaningful log messages about the inputs, during the execution of APIs. Based on the feedback from these log messages, the fuzzer is able to perform much more meaningful mutation than the random approach taken by Syzkaller. Take the WebGL program shown in Figure 4.2 as example, (1) when `bufferData` is executed with the first argument invalid and the browser emits a log message ("invalid target"), instead of mutating the input randomly, to effectively expand code coverage, the fuzzer should focus its mutation on the first argument; (2) Similarly, when `drawArrays` is called and the log message "no valid shader program in use" is emitted, we should add a call to `useProgram` before the current API to fix the internal WebGL program state. Thus the fuzzer avoids performing meaningless mutations on executed inputs and effectively reduce the search space.

4.4.2 Research Problems and Approaches

Ideally, given a log message reporting some error detected the input, we expect the fuzzer to mutate the specific part of the input that the message indicates in a smart way as a human analyst would take after reading the message, instead of taking random locations to mutate in a blind way.

To build such a smart fuzzer capable of performing meaningful mutations, the following research problems need to be addressed:

Q1. how to detect the type of log messages, i.e., type 1 or type 2?

Q2. If the log message indicates the invalidity of some argument passed to the identified API, which argument is it complaining about?

Q3. If the log message indicates the invalidity of a WebGL program internal state, which API to we use to fix the program state?

Q4. If all the previous questions are solved, given a log messages triggered at runtime, how to mutate the input to effectively expand coverage in the next iteration of fuzzing?

We studied the information available in the log messages emitted by browsers by manually executing some invalid WebGL programs randomly generated by our fuzzer. Though not standardized, in the implementation of browsers we tested (including Chrome, Firefox and Safari), the API name is emitted as one field of the log message. Thus using the value of this field, we can easily infer the API that emitted the log message. Additionally, the log message also contains an error type field[1], which directly indicates the invalidity of some argument (by field values such as "INVALID_ENUM" or "INVALID_VALUE") or the internal program state (by field value of "INVALID_OPERATION"). Thus the answer to Q1 is available in the log messages.

To address the remaining problems (Q2, Q3, Q4), one possible approach is to *manually* analyze the messages and encode mutating rules for each of the possible log messages. This approach turns out to be unscalable as the total number of log messages is huge, and creating of a mutating rule for a log message requires a fair amount of knowledge of the WebGL (or OpenGL) interface. Another plausible approach is to *automatically* infer which argument to mutate and/or how to perform the mutation using natural language analysis on the log message, e.g., by checking the name of the argument reported in the log message and how the argument is invalid. However, as our initial effort showed, this approach still needs a lot of manual effort to label the log messages as there is no ground facts to train tasks the data set, and naive keyword based matching based approaches resulted in very low precision.

[1]↑Note this field is not available in Firefox, but as we will show in next sections, we only use log messages from the Chrome browser.

In WEBGLFUZZER, we use a static analysis based approach to solve the these problems. Log messages are emitted by statement calling some specific routine (e.g., `SynthesizeGLError` in Chrome) defined in the browser source code. Such message logging statements are conditioned on checks on values originated from some arguments of the API or internal program state values. Based on this observation, we infer target arguments and dependent API set of log messages by analyzing the data dependencies of values of the checks that the message logging statements is conditioned on.

Identifying message logging statements and path condition collection. To infer the target arguments or dependent API set of some log message, the first step is to identify all the message logging statements and collect their path conditions. The problem of computation of all possible code paths from the API entry to message logging statements and collecting their path conditions is encoded as a standard IFDS (Inter-procedural Finite Distributive Subset) [98] data flow analysis problem. For each API, this step gives us a set of message logging statements and their path conditions. From the message logging statements, we extract the error type field (a constant argument passed to the message logging routine). According to the value of the error type field, we either infer the target arguments or the dependent API set as follows.

Inferring target arguments. If a log message indicates the invalidity of some argument (type 1), we infer its target argument by back tracking the value flow of the path conditions, if the value of some argument of the API under analysis flows into one of the path conditions, it is identified as a target argument (shown in Section 4.5.2). When the log message is triggered by the API at runtime, the fuzzer only mutate identified target arguments, instead of performing random mutation.

Inferring dependent API set. If a log message indicates the invalidity of the WebGL internal program state (of type 2), we infer the dependent API set of a log message by analyzing which APIs update the same fields of internal objects whose value flow into the path conditions (shown in Section 4.5.3). When the log message is triggered, the fuzzer will update the input based on the dependent API set in the hope of fixing the internal WebGL program state.

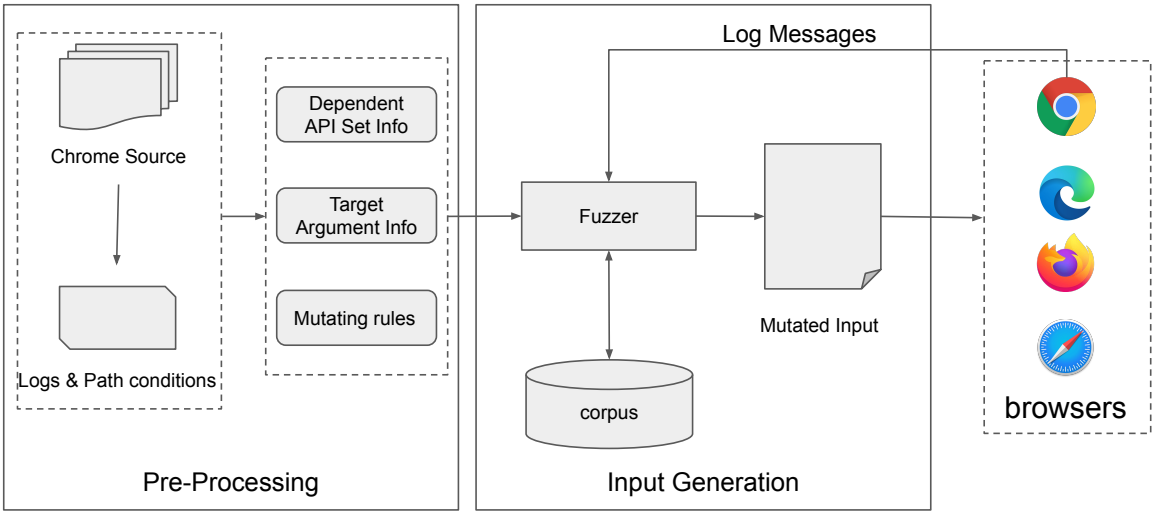


Figure 4.3. WEBGLFUZZER design overview. In the pre-processing stage, WEBGLFUZZER identifies all the log messages, their mutating rules, target argument and dependent API set, which is used by the fuzzer component to perform log message guided mutations on inputs.

4.5 Design

The goal of WEBGLFUZZER is to synthesize random WebGL programs (a sequence of WebGL API calls with prepared arguments) to test and detect anomalies in the WebGL implementation of browsers. Considering the difficulties in applying coverage guided fuzzing technique and its inherent limitations, we propose WEBGLFUZZER to mutate inputs under the guidance of runtime log messages. The core idea of WEBGLFUZZER is to leverage the log messages triggered during the execution WebGL APIs to guide the input mutation.

4.5.1 Workflow

Figure 4.3 depicts the overall workflow of WEBGLFUZZER. From a high level point of view, WEBGLFUZZER consists of a set of static analysis tools and a fuzzer component. The static analysis tools run in the pre-processing phase. Our analysis identifies all the log messages (message logging statements) and their path conditions in each API, and for each log message, computes the target arguments (see Section 4.5.2) and its mutating rule (see Section 4.5.4), if it is a message indicating the invalidity of some argument), or the dependent

API set (see Section 4.5.3), if the log message indicates invalidity of the internal WebGL program state.

At runtime in the fuzzing loop, WEBGLFUZZER takes the log messages from Chrome browser as feedback, and performs mutation on the input based on pre-computed by the static analysis tools (in pre-processing stage) information and the log messages (see Section 4.5.4).

4.5.2 Inferring Target Arguments

At runtime WEBGLFUZZER relies on pre-processed results of static analyzers to identify which arguments to mutate, when a log message indicating the invalidity of some arguments, is emitted by some API.

Our static analysis is based on the observation that log messages indicating the invalidity of some argument are emitted by statements conditioned on sanity checks on some value influenced by some argument of the API. For example, in the example shown in Listing 4.1, the "invalid target" message will be emitted by `ValidateBufferDataTarget` if its `target` argument, which originates from the type argument in `target` argument of `bufferData` method (the native implementation of WebGL API `bufferData`) , is detected to be invalid.

Algorithm 6 presents the algorithm for inferring the target arguments of a log message. Given a set of path conditions (computed using IFDS data flow analysis, see Section 4.4.2), and the source code of API under analysis, a static value flow graph is built following the algorithm introduced in [108] (line 1). A static value flow graph is a directed graph recording the def/use relationship of variables and expressions in the program, in which an edge $s \rightarrow d$ indicates s is used in d, or the value of s flows into d. Based on the value flow graph, for each of the path condition, we use a work list based algorithm to back trace the value flow graph from the path condition value. If some predecessor is an argument of the API under analysis, it is saved in the results (line 12).

Considering that the path conditions of one message logging statement may share some path conditions with other message logging statement, we analyze the path conditions in the order of distance from the message logging statement (by sorting the path conditions in line 2) and return the first non-empty result (line 15).

```
WebGLBuffer* ValidateBufferDataTarget(const char* function_name,
                                      GLenum target) {
  WebGLBuffer* buffer = nullptr;
  switch (target) {
    case GL_ELEMENT_ARRAY_BUFFER:
      buffer = bound_vertex_array_object_->BoundElementArrayBuffer();
      break;
    case GL_ARRAY_BUFFER:
      buffer = bound_array_buffer_.Get();
      break;
    default:
      SynthesizeGLError(GL_INVALID_ENUM, function_name, "invalid target");
      return nullptr;
  }
  if (!buffer) {
    SynthesizeGLError(GL_INVALID_OPERATION, function_name, "no buffer");
    return nullptr;
  }
  return buffer;
}
void BufferDataImpl(GLenum target, int64_t size, const void* data,
                    GLenum usage) {
  WebGLBuffer* buffer = ValidateBufferDataTarget("bufferData", target);
  if (!buffer)
    return;
  if (!ValidateBufferDataUsage("bufferData", usage))
    return;
  if (!ValidateValueFitNonNegInt32("bufferData", "size", size))
    return;
  buffer->SetSize(size);
  ContextGL()->BufferData(target, static_cast<GLsizeiptr>(size), data,
   usage);
}
void bufferData(GLenum target, int64_t size, GLenum usage) {
  if (isContextLost())
    return;
  BufferDataImpl(target, size, nullptr, usage);
}
```

Listing 4.1. Checks on WebGL arguments in Chrome browser

Algorithm 6: Inferring Target Arguments

Input: API: the API under analysis
Input: PCS: path conditions of some message logging statement
Output: $ARGS$: detected target arguments

```
1  VFG ← build_vfg(API)
2  sort PCS by distance from the message logging statement
3  for pc ∈ PCS do
4      work_list ← {pc}
5      visited ← {}
6      while work_list ≠ ∅ do
7          v ← work_list.pop()
8          if v ∉ visited then
9              for p ∈ v.pred do
10                 work_list ← {p} ∪ work_list
11             if v is an argument of API then
12                 ARGS ← ARGS ∪ {arg # of v}
13             visited ← {v} ∪ visited
14     if ARGS ≠ ∅ then
15         break
```

4.5.3 Inferring Dependent API Set

When an API emits a log message indicating the invalidity of the WebGL internal state, WEBGLFUZZER needs to choose an API from a set of dependent API set in the hope of fixing the program state.

Our analysis is based on the observation that log messages indicating the invalidity of the internal WebGL program state are emitted by some statements conditioned on checks on fields of some internal data structure, which are updated by the execution of some other APIs. For example, in the example shown in Listing 4.2, the statement emitting "no valid shader program in use" in `ValidateRenderingState` message is conditioned on *current_program_*, which is updated by `useProgram` API (line 43 and 44).

The dependent API set of a log message is computed in 3 steps. Firstly, we compute the list of internal variables that each API updates using the algorithm shown in Algorithm 7, which uses a work list based algorithm to traverse all the code paths to collect all internal

Algorithm 7: Collecting the list of update internal variables

Input: API: the API under analysis
Output: $UPDATES$: internal variables API updates

```
1  CFG ← build_cfg(API)
2  entry ← CFG.entry()
3  work_list ← {entry}
4  visited ← {}
5  while work_list ≠ ∅ do
6      v ← work_list.pop()
7      if v ∉ visited then
8          for s ∈ v.succ do
9              work_list ← {s} ∪ work_list
10         if v updates an internal variable then
11             UPDATES ← UPDATES ∪ get_updates(v)
12         visited ← {v} ∪ visited
```

variables an API updates. Here we use a type-based approach to identify internal variables of WebGL: all variables of types defined in the WebGL namespace or fields of such variables are considered as a WebGL internal variable. In the second step, we collect the internal variable the path conditions of a message logging statement depend on, using a similar algorithm as Algorithm 6. Lastly, given a log message M with a set of dependencies on internal variables S_d and an API API with a set of internal variables (S_u)it updates, if S_d overlap with S_u (i.e., $S_d \wedge S_u \neq \varnothing$), API is added to the dependent API set of M.

4.5.4 Log Guided Mutation

At runtime, if a log message is triggered by an API, depending of the error type of the log message, WEBGLFUZZER either performs mutation of the target arguments of the API or tries to fix the internal program states using the dependent API set.

Mutating target arguments. Log messages also contain rich semantic information regarding the current argument value. We can leverage such semantic information when performing mutations on the arguments. E.g., if an argument of `enum` type is complained to be invalid, we should avoid generating the same value for the argument; if some numeric

```
bool ValidateRenderingState(const char* function_name) {
  // Command buffer will not error if no program is bound.
  if (!current_program_) {
    SynthesizeGLError(GL_INVALID_OPERATION, function_name,
                      "no valid shader program in use");
    return false;
  }
  return true;
}
bool ValidateDrawArrays(const char* function_name) {
  if (isContextLost())
    return false;
  if (!ValidateRenderingState(function_name)) {
    return false;
  }
  const char* reason = "framebuffer incomplete";
  if (framebuffer_binding_ && framebuffer_binding_->
   CheckDepthStencilStatus(
    &reason) != GL_FRAMEBUFFER_COMPLETE) {
      SynthesizeGLError(GL_INVALID_FRAMEBUFFER_OPERATION, function_name,
   reason);
      return false;
  }
  return true;
}
void drawArrays(GLenum mode, GLint first, GLsizei count) {
  if (!ValidateDrawArrays("drawArrays"))
    return;
  if (!bound_vertex_array_object_->IsAllEnabledAttribBufferBound()) {
    SynthesizeGLError(GL_INVALID_OPERATION, "drawArrays",
                      "no buffer is bound to enabled attribute");
    return;
  }
  // ...
}
void useProgram(WebGLProgram* program) {
  if (!ValidateNullableWebGLObject("useProgram", program))
    return;
  if (program && !program->LinkStatus(this)) {
    SynthesizeGLError(GL_INVALID_OPERATION, "useProgram", "program not
   valid");
    return;
  }
  if (current_program_ != program) {
    if (current_program_)
      current_program_->OnDetached(ContextGL());
    current_program_ = program;
    ContextGL()->UseProgram(ObjectOrZero(program));
    if (program)
      program->OnAttached();
  }
}
```

Listing 4.2. Checks on WebGL internal program state in Chrome browser

value is complained to be too large or too small, we should choose a smaller one or larger value. Such information is easy for a human analyst to understand, however, challenging to be understood programmatically. In the current stage, we studied the log messages and manually build a set of **meaningful mutation** as the previous examples show. At runtime we use a keyword matching based approach to infer the semantic meaning of log messages, apply meaningful mutations according to the detected semantic meaning.

Fixing program states. When the fuzzer sees a log message indicating the invalidity of the WebGL program internal state, it means that the some internal variables are not in the expected state. Such internal variables are updated by the dependent API set we compute in Section 4.5.3. To mutate the input (the WebGL program), there are a few cases to consider. (1) One of the APIs from the dependent API set has been called as some point before the current API; if its previous execution succeeded (without triggering a log message), we either remove the dependent API call in the input and add another API from the dependent API set to the program, if the execution of the dependent API failed (with some log message triggered), the current API is not handled as the internal state may have been updated when handling the dependent API. (2) if no dependent APIs are called before the current API, we add a call to one of its dependent APIs before the current API call.

4.5.5 Multi-browser Log Guided Fuzzing

Log guided fuzzing we propose in this work relies on static analysis on the source code of WebGL implementation and collection of log messages at runtime. Because of these requirements, it is difficult to apply log guided fuzzing on close-sourced browsers (e.g., Safari) and run time log message collection is infeasible without non-trivial engineering work (e.g., Firefox).

Considering that WebGL implementations in different browsers follow the same specification. To apply log guided fuzzing in close-sourced browsers and browsers where runtime log message collection is infeasible, WEBGLFUZZER uses a multi-browser execution technique. More specifically, the fuzzer tests the inputs on multiple browsers, but its input mutation is based on only one browser amenable to static analysis and runtime log message collection.

As a result, WEBGLFUZZER can fuzz WebGL implementations in one browser based on analysis of another browser.

In WEBGLFUZZER, all the static analysis is performed on the source code in Chrome, thus the log guided fuzzing is built based on the analysis on Chrome. With mutli-browser fuzzing technique, WEBGLFUZZER is able to test other popular browsers (Edge, Firefox and Safari).

4.6 Implementation

4.6.1 Static Analysis

The static analysis is based on LLVM IR. To this end, we modified the build system of Chrome to generate LLVM bitcode using wllvm [95].

The WebGL implementation of Chrome contains C++ virtual call sites whose target is unknown statically. This leads to challenges in control flow graph construction, which is a common task other static analysis depends on. We make the following observations regarding the generated LLVM bit code for virtual calls: (1) the function pointer is loaded from a table located at the beginning of an object, using a constant offset ($Offset$); (2) the first argument (this pointer) of the call site is the same as the object where the function pointer is loaded from. Based on these observations, we resolve the class of the virtual call site to the type of the first argument (C), the targets to be the set of methods at the same offset of the vtables of the class C and its subclasses.

Our analysis for identifying all the message logging statements and collecting the path conditions is formulated as an IFDS problem [98] in the phasar [84] framework. The implementation of the problem collects the operands of branch or switch instructions in the code paths, terminates the analysis along the current code path when a call instruction to the message logging function is detected. To compute the static value flow in the implementation of WebGL APIs, SVF [87] is used to build a value flow graph used in Algorithm 6.

4.6.2 Test Execution

At runtime, WEBGLFUZZER detects anomalies by testing each the generated WebGL programs (input) in browsers. In WEBGLFUZZER, tests are transmitted to a test executor component running in the browsers for execution. The test executor is implemented as a web page containing a Javascript routine for parsing and executing WebGL programs received from the fuzzer. The test executor is loaded when the browsers under tests when they are started.

In implementation, WEBGLFUZZER communicates with browser through the standard WebDriver [126] interface using Selenium [85]. To test browser on Android and iOS devices, Appium [82] is used as a proxy between the the fuzzer and browsers running on the device side.

4.7 Evaluation

4.7.1 Experimental Setup

We are currently evaluating popular browsers on both desktop and mobile OSes. Table 4.2 lists our experimenal setup. Note, due to lack of support from Appium [82] (the tool we use to test mobile browsers in Android and iOS), on Android only Chrome, and on iOS only Safari is tested.

4.7.2 Bug Finding

Table 4.3 shows the list of bugs we have found so far, including 3 bugs in Chrome, 1 bug in Firefox, and 2 bugs in Safari. Out of these 6 bugs, 2 are resulted from GPU hang, one (marked with *) even causes X-Server freeze, and 2 cause tab crash, potentially because of memory related bugs.

4.8 Conclusion

WebGL interface exposes the underlying graphics stack to remote attacks. Many recent high severity CVEs in poppular browsers like Chrome and Firefox demonstrate its security

Table 4.2. WEBGLFUZZER Experimental Setup

OS	Browsers	GPU
Windows	Chrome	Intel
	Firefox	
	Edge	
	Opera	
Linux	Chrome	Intel, NVIDIA Tegra X1
	Firefox	
MacOS	Chrome	Intel
	Firefiox	
	Safari	
	Edge	
Android	Chrome	Qualcomm Adreno
iOS	Safari	Apple GPU

Table 4.3. the list of bugs found WEBGLFUZZER

Summary	Browser	OS	GPU
WebGL tab crash and system wide OOM	Firefox	MacOS, Linux	Intel
GPU Hang (*)	Chrome	Linux	Intel
Assertion failure	Chrome	Linux	Intel
Assertion Failure	Chrome	Linux	Intel
Tab crash	Safari	MacOS	Intel
GPU Hang	Safari	iOS	Apple GPU

risks (e.g., CVE-2020-6492 [73] or CVE-2020-15675 [72]). To proactively address the security risks of WebGL, we propose a fuzzing approach to uncover remotely triggerable bugs. However, it is difficult to apply state-of-art coverage guided fuzzing technique to this target because of its inherent dependency on code coverage collection which is challenging when analyzing the WebGL stack, because the software stack consists of several layers of libraries, different processes, and code running both in user and kernel space. In addition, several of the code blocks are closed source and proprietary. Even with precise coverage feedback, it does not indicate how to effectively extend coverage when performing mutation on the current input.

We propose a novel technique, called log message guided fuzzing, which eliminates dependency on code coverage to perform meaningful mutations. To build a log message guided

fuzzer, for log messages indicating the invalidity of arguments, we identify the target arguments, and buid a set of mutating rules from the the semantic meaning of the log message using natural language analysis; for log messages indicating invalidity of internal program state, we compute its a dependent API set. Using these information, instead of performing random mutation, the fuzzer focuses on mutating the identified target argument using mutating rules built on semantic analysis on log message, or tries to fix the internal program state using APIs from its dependent set.

So far, our evaluation in popular browsers (Chrome, Firefox and Safari) on both desktop (Linux, Windows and MacOS) and mobile (Andrdoid and iOS) OSes has led to the discovery of 6 bugsx, 3 in Chrome, 2 in Safari, and 1 in Firefox, one of the bugs in Chrome freezes the X-Server in Linux.

5. SUMMARY

State-of-art coverage guided fuzzing technique faces challenges when applied in real-world systems. In particular, based on our study and experiments we identify the following limitations in existing coverage guided fuzzers: (1) the *coverage wall issue*, fuzzer-generated inputs cannot bypass complex sanity checks in the target programs and are unable to cover code paths protected by such checks; (2) *inability to adapt to interfaces* to inject fuzzer-generated inputs, one important example of such interface is the software/hardware interface between drivers and their devices; (3) *dependency on code coverage feedback*, this dependency makes it hard to apply fuzzing to targets where code coverage collection is challenging (due to proprietary components or special software design).

To address the coverage wall issue, we propose T-FUZZ, a program transformation based approach to address coverage wall issue. Whenever the coverage wall is reached, T-FUZZ identifies and removes the hard-to-bypass checks from the target program. Fuzzing then continues on the transformed program, allowing the code protected by the removed checks to be triggered and potential bugs discovered. To address the false positives caused by program transformation, T-FUZZ leverages a symbolic execution-based approach to filter out false positives and reproduce true bugs in the original program.

Our evaluation shows that T-FUZZ outperforms existing technique: on the CGC dataset, T-FUZZ finds bugs in 166 binaries, while Driller found bugs in 121, and AFL only in 105 binaries. T-FUZZ also found 4 new bugs in previously-fuzzed programs and libraries.

To address the inability to adapt to inferfaces, we propose USBFUZZ which targets fuzzing the software/hardware interface in the USB stack. In its core, USBFUZZ emulates an special USB device that provides data to the device driver (when it performs IO operations). This allows us to fuzz the input space of drivers from the device's perspective, an angle that is difficult to achieve with real hardware. USBFUZZ discovered 53 bugs in Linux (out of which 37 are new, and 36 are memory bugs of high security impact, potentially allowing arbitrary read or write in the kernel address space), one bug in FreeBSD, four bugs (resulting in Blue Screens of Death) in Windows and three bugs (two causing an unplanned restart, one freezing the system) in MacOS.

To break the dependency on code coverage feedback, we propose WEBGLFUZZER. To fuzz the WebGL interface, where code coverage collection is challenging, we introduce WEBGLFUZZER, which internally uses a log guided fuzzing technique. WEBGLFUZZER is not dependent on code coverage feedback, but instead, makes use of the log messages emitted by browsers to guide its input mutation. Compared with coverage guided fuzzing, our log guided fuzzing technique is able to perform more meaningful mutation under the guidance of the log message. WEBGLFUZZER is under evaluation and so far, it has found 6 bugs, one of which is able to freeze the X-Server.

REFERENCES

[1] afl-lafintel. *Circumventing Fuzzing Roadblocks with Compiler Transformations*. [Online; accessed 16-September-2017]. 2017. URL: https://lafintel.wordpress.com/.

[2] Dave Aitel. "An introduction to SPIKE, The fuzzer creation kit". In: *presentation slides), Aug* 1 (2002).

[3] Sebastian Angel et al. "Defending against Malicious Peripherals with Cinch." In: *USENIX Security Symposium*. 2016, pp. 397–414.

[4] MIDI Association. *Basics of USB-MIDI*. [Online; accessed 15-Nov-2018]. 2018. URL: https://www.midi.org/articles-old/basic-of-usb.

[5] SD Association. *SD Standard overview*. https://www.sdcard.org/developers/overview/. 2020.

[6] Thanassis Avgerinos et al. "Enhancing symbolic execution with veritesting". In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 1083–1094.

[7] Tiffany Bao et al. "Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits". In: *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE. 2017, pp. 824–839.

[8] Adam Barth et al. "The security architecture of the chromium browser". In: *Technical report*. Stanford University, 2008.

[9] knowledge base. *ar5212*. https://whirlpool.net.au/wiki/ar5212.

[10] Osbert Bastani et al. "Synthesizing Program Input Grammars". In: *PLDI 2017*. Barcelona, Spain: ACM, 2017, pp. 95–110. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062349. URL: http://doi.acm.org/10.1145/3062341.3062349.

[11] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. 2005.

[12] Silas Boyd-Wickizer and Nickolai Zeldovich. "Tolerating Malicious Device Drivers in Linux." In: *USENIX annual technical conference*. Boston. 2010.

[13] Hartley Brody. *USB Rubber Ducky Tutorial: The Missing Quickstart Guide to Running Your First Keystroke Payload Hack*. https://blog.hartleybrody.com/rubber-ducky-guide/. 2017.

[14] David Brumley et al. “Automatic patch-based exploit generation is possible: Techniques and implications”. In: *SP’08*. IEEE. 2008, pp. 143–157.

[15] bugzilla. *Bug 104798 - endless loop resulting OOM*. [Online; accessed 26-Jan-2018]. 2018. URL: .

[16] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.” In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[17] Microsoft Security Response Center. *WebGL Considered Harmful*. https://msrc-blog.microsoft.com/2011/06/16/webgl-considered-harmful/. 2021.

[18] Sang Kil Cha, Maverick Woo, and David Brumley. “Program-adaptive mutational fuzzing”. In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pp. 725–741.

[19] Jiongyi Chen et al. “IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing”. In: *NDSS*. 2018.

[20] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. “S2E: A platform for in-vivo multi-path analysis of software systems”. In: *Acm Sigplan Notices* 46.3 (2011), pp. 265–278.

[21] Catalin Cimpanu. *List of 29 Different Types of USB Attacks*. https://www.bleepingcomputer.com/news/security/heres-a-list-of-29-different-types-of-usb-attacks/. 2019.

[22] Android Developers. *WebView — Android Developers*. urlhttps://developer.android.com/referer 2021.

[23] die.net. *lsusb(8) - Linux man page*. https://linux.die.net/man/8/lsusb. 2018.

[24] Linux kernel document. *KernelAddressSanitizer*. [Online; accessed 20-Feb-2018]. 2018. URL: https://github.com/google/kasan/wiki.

[25] Linux kernel document. *The Kernel Address Sanitizer (KASAN)*. [Online; accessed 20-Feb-2018]. 2018. URL: https://www.kernel.org/doc/html/v4.12/dev-tools/kasan.html.

[26] Apple Developer Documentation. *WKWebView — Apple Developer Documentation*. urlhttps://developer.apple.com/documentation/webkit/wkwebview. 2021.

[27] B. Dolan-Gavitt et al. “LAVA: Large-Scale Automated Vulnerability Addition”. In: *SP’16)*. May 2016, pp. 110–121. DOI: 10.1109/SP.2016.15.

[28] Will Drewry and Tavis Ormandy. “Flayer: Exposing Application Internals.” In: *WOOT* 7 (2007), pp. 1–9.

[29] Michael Eddington. “Peach fuzzing platform”. In: *Peach Fuzzer* (2011), p. 34.

[30] Electron. *Electron — Build cross-platform desktop apps with JavaScript, HTML, and CSS*. urlhttps://www.electronjs.org/. 2021.

[31] Ellisys. *Explorer 200 - Hardware trigger*. https://www.ellisys.com/products/usbex200/trigger.php.

[32] Jesse Hertz etc. *Project Triforce: AFL + QEMU + kernel = CVEs! (or) How to use AFL to fuzz arbitrary VMs.* . 2018.

[33] Sergiu Gatlan. *Google Chrome 85 fixes WebGL code execution vulnerability*. https://www.bleepingcomputer.com/news/security/google-chrome-85-fixes-webgl-code-execution-vulnerability/. 2020.

[34] Github. *Dummy/Loopback USB host and device emulator driver*. [Online; accessed 15-Nov-2018]. 2018. URL: .

[35] github. *tracer: Utilities for generating dynamic traces*. [Online; accessed 21-Oct-2017]. 2017. URL: https://github.com/angr/tracer.

[36] Patrice Godefroid. “From blackbox fuzzing to whitebox fuzzing towards verification”. In: *Presentation at the 2010 International Symposium on Software Testing and Analysis*. 2010.

[37] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: directed automated random testing”. In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 213–223.

[38] Patrice Godefroid, Michael Y Levin, and David Molnar. “SAGE: whitebox fuzzing for security testing”. In: *Queue* 10.1 (2012), p. 20.

[39] Patrice Godefroid, Hila Peleg, and Rishabh Singh. “Learn&Fuzz: Machine Learning for Input Fuzzing”. In: *CoRR* abs/1701.07232 (2017). arXiv: 1701.07232. URL: http://arxiv.org/abs/1701.07232.

[40] Enes Göktas et al. “Out of control: Overcoming control-flow integrity”. In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 575–589.

[41] GoodFET. *GoodFET-Facedancer21*. [Online; accessed 20-Feb-2018]. 2018. URL: http://goodfet.sourceforge.net/hardware/facedancer21/.

[42] Google. *External USB fuzzing for Linux kernel*. [Online; accessed 20-Feb-2018]. 2018. URL:
.

[43] Google. *Found Linux kernel USB bugs*. [Online; accessed 20-Feb-2018]. 2018. URL:
.

[44] Google. *HonggFuzz*. [Online; accessed 10-April-2017]. 2017. URL: https://google.github.io/honggfuzz/.

[45] Google. *KASAN-Linux usb-fuzzer*. https://github.com/google/kasan/tree/usb-fuzzer. 2020.

[46] Google. *OSS-Fuzz - Continuous Fuzzing for Open Source Software*. [Online; accessed 10-April-2017]. 2016. URL: https://github.com/google/oss-fuzz.

[47] Google. *Syzkaller - kernel fuzzer*. [Online; accessed 20-Feb-2018]. 2018. URL: https://github.com/google/syzkaller.

[48] Khronos Group. *OpenGL ES Overview*. urlhttps://www.khronos.org/opengles/. 2021.

[49] Khronos Group. *WebGL Overview*. urlhttps://www.khronos.org/webgl/. 2021.

[50] NCC Group. *AFL/QEMU fuzzing with full-system emulation.* [Online; accessed 20-Feb-2018]. 2018. URL: https://github.com/nccgroup/TriforceAFL.

[51] NCC Group. *Umap2*. https://github.com/nccgroup/umap2.

[52] Shellphish Group. *Driller: augmenting AFL with symbolic execution!* [Online; accessed 20-Septempber-2017]. 2017. URL: https://github.com/shellphish/fuzzer.

[53] Istvan Haller et al. "Dowser: a guided fuzzer to find buffer overflow vulnerabilities". In: *SEC'13*. 2013, pp. 49–64.

[54] HyungSeok Han and Sang Kil Cha. "IMF: Inferred model-based fuzzer". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2345–2358.

[55] Intel. *Intel Virtualization Technology*. [Online; accessed 20-Oct-2018]. 2018. URL: https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html.

[56] Dave Jones. *trinity:Linux system call fuzzer*. [Online; accessed 20-Feb-2018]. 2018. URL: https://github.com/kernelslacker/trinity.

[57] Samy Kamkar. *USBdriveby: Exploiting USB in style*. http://samy.pl/usbdriveby/. 2014.

[58] Ulf Kargén and Nahid Shahmehri. "Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 782–792.

[59] Sylvester Keil and Clemens Kolbitsch. "Stateful fuzzing of wireless device drivers in an emulated environment". In: *Black Hat Japan* (2007).

[60] Mustakimur Rahman Khandaker et al. "Origin-sensitive control flow integrity". In: *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 2019, pp. 195–211.

[61] David Kierznowski. *BadUSB 2.0: Exploring USB Man-In-The-Middle Attacks*. 2015.

[62] Andrey Konovalov. *CVE-2016-2384: Exploiting a double-free in the USB-MIDI Linux kernel driver*. [Online; accessed 15-Nov-2018]. 2016. URL: https://xairy.github.io/blog/2016/cve-2016-2384.

[63] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. "Testing closed-source binary device drivers with DDT". In: *USENIX Annual Technical Conference*. 2010.

[64] caca labs. *zzuf - multi-purpose fuzzer*. [Online; accessed 10-October-2017]. 2017. URL: http://caca.zoy.org/wiki/zzuf.

[65] Jon Larimer. *Beyond Autorun: Exploiting vulnerabilities with removable storage*.

. 2011.

[66] Yuekang Li et al. "Steelix: program-state based binary fuzzing". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pp. 627–637.

[67] LLC Lunge Technology. *CGC Corpus*. [Online; accessed 20-September-2017]. 2017. URL: http://www.lungetech.com/2017/04/24/cgc-corpus/.

[68] LXR. *Linux Source Code*. https://elixir.bootlin.com/linux/latest/source/drivers/media/mc/mc-entity.c#L666. 2020.

[69] Richard McNally et al. *Fuzzing: the state of the art*. Tech. rep. DEFENCE SCIENCE and TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), 2012.

[70] Microsoft. *LifeCam VX-800*. https://www.microsoft.com/accessories/en-us/d/lifecam-vx-800. 2018.

[71] Microsoft. *Microsoft security development lifecycle*. [Online; accessed 10-April-2017]. 2017. URL: https://www.microsoft.com/en-us/sdl/process/verification.aspx.

[72] MITRE. *CVE-2020-15675*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15675. 2021.

[73] MITRE. *CVE-2020-6492*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-6492. 2021.

[74] David Molnar, Xue Cong Li, and David Wagner. "Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs." In: *SEC*. Vol. 9. 2009, pp. 67–82.

[75] Mozilla. *Mozilla Foundation Security Advisory 2020-42*. https://www.mozilla.org/en-US/security/advisories/mfsa2020-42/. 2020.

[76] Matthias Neugschwandtner et al. "The BORG: Nanoprobing binaries for buffer overreads". In: *SP'15*. ACM. 2015, pp. 87–97.

[77] Nir Nissim, Ran Yahalom, and Yuval Elovici. "USB-based attacks". In: *Computers & Security* 70 (2017), pp. 675–688.

[78] NVD. *Common Vulnerabilities and Exposures:CVE-2016-2384*. [Online; accessed 15-Nov-2018]. 2018. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2384.

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

[80] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. "POTUS: Probing Off-The-Shelf USB Drivers with Symbolic Fault Injection". In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. 2017.

[81] Matt Porter. *Kernel USB Gadget Configfs Interface*. [Online; accessed 15-Nov-2018]. 2018. URL:

.

[82] Appium project. *Appium: Mobile App Automation Mad e Awesome.* urlhttp://appium.io/. 2021.

[83] Chromium Project. *ANGLE - Almost Native Graphics Layer Engine*. urlhttps://chromium.goog 2021.

[84] PhASAR Project. *phasar: an LLVM-based static analysis framework*. https://phasar.org/. 2021.

[85] Selenium Project. *SeleniumHQ browser automation*. https://www.selenium.dev/. 2021.

[86] Spice Project. *usbredir*. [Online; accessed 20-Feb-2018]. 2018. URL: https://www.spice-space.org/usbredir.html.

[87] SVF Project. *Static Value-Flow Analysis Framework for Source Code*. https://github.com/SVF-tools/SVF. 2021.

[88] Syzkaller Project. *report.go*. https://github.com/google/syzkaller/blob/master/pkg/report/report.go.

[89] The Cordava Project. *Apache CORDAVA*. https://cordova.apache.org/. 2021.

[90] USB/IP Project. *USB/IP PROJECT*. http://usbip.sourceforge.net/.

[91] USBGuard project. *USB Guard*. https://usbguard.github.io/. 2018.

[92] The Chromium Projects. *Chromium OS - The Chromium Projects*. urlhttps://www.khronos.org 2021.

[93] The Chromium Projects. *Control Flow Integrity*. https://www.chromium.org/developers/testing/control-flow-integrity. 2021.

[94] David A Ramos and Dawson R Engler. "Under-Constrained Symbolic Execution: Correctness Checking for Real Code." In: *USENIX Security Symposium*. 2015, pp. 49–64.

[95] Tristan Ravitch. *wllvm: A wrapper script to build whole-program LLVM bitcode files*. https://github.com/travitch/whole-program-llvm. 2021.

[96] Sanjay Rawat et al. “Vuzzer: Application-aware evolutionary fuzzing”. In: *NDSS*. 2017.

[97] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. “SymDrive: testing drivers without devices”. In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 2012, pp. 279–292.

[98] Thomas Reps, Susan Horwitz, and Mooly Sagiv. “Precise interprocedural dataflow analysis via graph reachability”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1995, pp. 49–61.

[99] J Röning et al. “Protos-systematic approach to eliminate software vulnerabilities”. In: *Invited presentation at Microsoft Research* (2002).

[100] Sergej Schumilo, Ralf Spenneberg, and Hendrik Schwartke. “Don’t trust your USB! How to find bugs in USB device drivers”. In: *Blackhat Europe* (2014).

[101] Sergej Schumilo et al. “kafl: Hardware-assisted feedback fuzzing for OS kernels”. In: *Adresse: https://www. usenix. org/system/files/conference/usenixsecurity17/sec17-schumilo. pdf (besucht am 10. 08. 2017)*. 2017.

[102] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C”. In: *ESEC/FSE-13*. Lisbon, Portugal: ACM, 2005, pp. 263–272. ISBN: 1-59593-014-0. DOI: 10.1145/1081706.1081750. URL: http://doi.acm.org/10.1145/1081706.1081750.

[103] Y. Shoshitaishvili et al. “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *SP’16*. May 2016, pp. 138–157. DOI: 10.1109/SP.2016.17.

[104] Adam Slowik and Halina Kwasnicka. “Evolutionary algorithms and their applications to engineering problems”. In: *Neural Computing and Applications* 32.16 (Mar. 2020), pp. 12363–12379. DOI: 10.1007/s00521-020-04832-8. URL: https://doi.org/10.1007/s00521-020-04832-8.

[105] Dokyung Song et al. “PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary”. In: *2019 Network and Distributed Systems Security Symposium (NDSS)*. Internet Society. 2019.

[106] Nick Stephens et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution.” In: *NDSS*. Vol. 16. 2016, pp. 1–16.

[107] Alan Stern. *Dummy/Loopback USB host and device emulator driver*. [Online; accessed 15-Nov-2018]. 2018. URL:

.

[108] Yulei Sui and Jingling Xue. “SVF: interprocedural static value-flow analysis in LLVM”. In: *Proceedings of the 25th international conference on compiler construction*. 2016, pp. 265–266.

[109] honggfuzz team. *Trophies (honggfuzz)*. [Online; accessed 4-Jan-2019]. 2019. URL: https://github.com/google/honggfuzz#trophies.

[110] ImageMagick team. *assertion error in RelinquishMagickResource*. [Online; accessed 23-Jan-2018]. 2018. URL: https://github.com/ImageMagick/ImageMagick/issues/955.

[111] LLVM team. *libFuzzer – a library for coverage-guided fuzz testing*. [Online; accessed 28-Jan-2018]. 2018. URL: https://llvm.org/docs/LibFuzzer.html.

[112] Qemu Team. *QEMU: the FAST! processor emulator*. [Online; accessed 10-Nov-2017]. 2018. URL: https://www.qemu.org/.

[113] QEMU team. *Device Specification for Inter-VM shared memory device*. [Online; accessed 10-Nov-2018]. 2018. URL: https://github.com/qemu/qemu/blob/master/docs/specs/ivshmem-spec.txt.

[114] radare2 team. *radare2: unix-like reverse engineering framework and commandline tools*. [Online; accessed 21-Oct-2017]. 2017. URL: https://github.com/radare/radare2.

[115] syzkaller team. *vusb ids*.
. 2018.

[116] syzkaller team. *vusb ids*. [Online; accessed 20-Feb-2018]. 2018. URL:
.

[117] syzkaller team. *vusb.txt at google/syskaller*. https://github.com/google/syzkaller/blob/master/sys/linux/vusb.txt. 2018.

[118] Dave Jing Tian et al. “LBM: A Security Framework for Peripherals within the Linux Kernel”. In: *LBM: A Security Framework for Peripherals within the Linux Kernel*. 2019, p. 0.

[119] Dave Jing Tian et al. “Making USB Great Again with USBFILTER”. In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 415–430.

[120] Jing Tian et al. “SoK:” Plug & Pray” Today–Understanding USB Insecurity in Versions 1 Through C”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 1032–1047.

[121] Linus Torvalds. *Kernel parameters*. [Online; accessed 20-Feb-2018]. 2018. URL: https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/kernel-parameters.txt.

[122] Ubuntu. *Package: streamer (3.103-3build1)*. https://packages.ubuntu.com/xenial/streamer. 2019.

[123] usbkill.org. *Official USB Killer Site*. https://usbkill.com/. 2019.

[124] Rijnard Van Tonder and Herman A Engelbrecht. "Lowering the USB Fuzzing Barrier by Transparent Two-Way Emulation." In: *WOOT*. 2014.

[125] Dmitry Vyukov. *kernel: add kcov code coverage*. [Online; accessed 20-Oct-2018]. 2018. URL: https://lwn.net/Articles/671640/.

[126] W3C. *WebDriver-W3C Working Draft 24 August 2020*. https://www.w3.org/TR/webdriver/. 2021.

[127] Junjie Wang et al. *Skyfire: Data-driven seed generation for fuzzing*. 2017.

[128] Tielei Wang et al. "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection". In: *SP'10*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 497–512. ISBN: 978-0-7695-4035-1. DOI: 10.1109/SP.2010.37. URL: http://dx.doi.org/10.1109/SP.2010.37.

[129] Wikipedia. *Extensible Host Controller Interface*. . 2018.

[130] Wikipedia. *Wireless USB*. . 2019.

[131] wikipedia. *USB On-The-Go*. [Online; accessed 20-Oct-2018]. 2018. URL: .

[132] Zhihao Yao et al. "Milkomeda: Safeguarding the Mobile GPU Interface Using WebGL Security Checks". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 1455–1469.

[133] Jonas Zaddach et al. "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares." In: *NDSS*. 2014.

[134] Michal Zalewski. *afl-fuzz: making up grammar with a dictionary in hand*. [Online; accessed 16-September-2017]. 2015. URL: https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html.

[135] Michal Zalewski. *american fuzzy lop*. [Online; accessed 1-August-2017]. 2017. URL: http://lcamtuf.coredump.cx/afl/.

[136] Michal Zalewski. *The bug-o-rama trophy case*. [Online; accessed 20-Septempber-2017]. 2017. URL: http://lcamtuf.coredump.cx/afl/#bugs.