Beyond the Coverage Plateau: A Comprehensive Study of Fuzz Blockers (Registered Report)

Wentao Gao
wentaog1@student.unimelb.edu.au
The University of Melbourne
Melbourne, Australia

Van-Thuan Pham
thuan.pham@unimelb.edu.au
The University of Melbourne
Melbourne, Australia

Dongge Liu
donggeliu@google.com
Google
Sydney, Australia

Oliver Chang
ochang@google.com
Google
Sydney, Australia

Toby Murray
toby.murray@unimelb.edu.au
The University of Melbourne
Melbourne, Australia

Benjamin I.P. Rubinstein
benjamin.rubinstein@unimelb.edu.au
The University of Melbourne
Melbourne, Australia

ABSTRACT

Fuzzing and particularly code coverage-guided greybox fuzzing is highly successful in automated vulnerability discovery, as evidenced by the multitude of vulnerabilities uncovered in real-world software systems. However, results on large benchmarks such as FuzzBench indicate that the state-of-the-art fuzzers often reach a plateau after a certain period, typically around 12 hours. With the aid of the newly introduced FuzzIntrospector platform, this study aims to analyze and categorize the fuzz blockers that impede the progress of fuzzers. Such insights can shed light on future fuzzing research, suggesting areas that require further attention. Our preliminary findings reveal that the majority of top fuzz blockers are not directly related to the program input, emphasizing the need for enhanced techniques in automated fuzz driver generation and modification.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software libraries and repositories.*

KEYWORDS

fuzzing, vulnerability detection, software security

ACM Reference Format:
Wentao Gao, Van-Thuan Pham, Dongge Liu, Oliver Chang, Toby Murray, and Benjamin I.P. Rubinstein. 2023. Beyond the Coverage Plateau: A Comprehensive Study of Fuzz Blockers (Registered Report). In *Proceedings of the 2nd International Fuzzing Workshop (FUZZING '23), July 17, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3605157.3605177

1 INTRODUCTION

Fuzzing, specifically Coverage-Guided Greybox Fuzzing (CGF), has received significant attention from both industry and academia in recent years due to its simplicity and high performance in automated vulnerability discovery. Popular CGF Fuzzers such as libFuzzer [8], AFL/AFL++ [1, 28], and Honggfuzz [6] have discovered thousands of vulnerabilities in large real-world systems [3, 21, 36].

The state-of-the-art in fuzzing has seen significant advancements, with hundreds of research papers and dozens of tools being published to improve the technique in various aspects [36]. These efforts have focused on enhancing fuzzing in areas such as feedback collection [16, 25, 27, 30], corpus management [29], seed selection algorithms [22, 23], input generation algorithms [15, 20, 31, 44, 47], and novel test oracle designs [39, 45]. Additionally, researchers have attempted to extend the applicability of fuzzing to challenging targets such as network protocols [18, 43], database systems [45, 50], SMT solvers [40], compilers [26], device drivers [41], and heterogeneous applications [48]. Another noteworthy research direction is parallel or distributed fuzzing [33, 38, 42], which aims to improve fuzzing efficiency by utilizing high-performance computing resources.

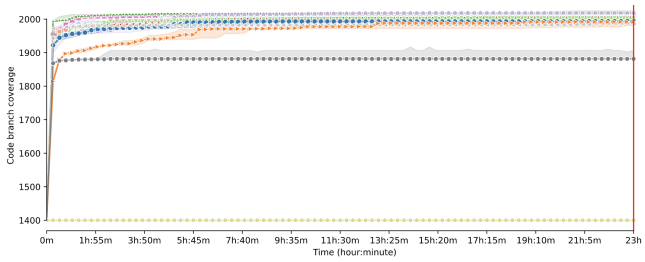

Figure 1: SBFT'23 Fuzzing Competition [35] result of LibPNG. Mean branch coverage growth over time is reported. At least 20 trials/fuzzer, 23 hours per trial. Most fuzzers reach their plateau within 14 hours.

Despite these advancements, the top-performing fuzzers continue to exhibit limitations as evidenced by results obtained from FuzzBench [37] and FuzzIntrospector [5]. As shown in Figure 1, all fuzzers participated in the recent SBFT Fuzzing Competition [35] reached their plateau after testing the popular LibPNG library [9] for 14 hours, with little to no further improvement in code coverage. We have observed similar trajectories in other benchmarked programs. *We need a better understanding of why this happens.*

FUZZING '23, July 17, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0247-1/23/07.
https://doi.org/10.1145/3605157.3605177

This study aims to conduct a comprehensive investigation of well-tested subject programs to identify and classify the fuzz blockers that impede the progress of fuzzers. The results of our analysis could serve as a valuable resource for future fuzzing research, providing guidance on how to (i) develop innovative approaches to address previously unknown types of blockers, (ii) allocate more resources towards important but under-explored research areas (e.g., automated fuzz driver generation [19, 32], configuration fuzzing [49]), or (iii) re-evaluate and re-design existing solutions for well-studied blockers such as magic numbers and checksums if they still persist.

To achieve the generality of our study findings, we have established a set of selection criteria to choose subject libraries/programs based on factors such as their popularity, their code size, their diversity in application domains, and the number of existing fuzz drivers. Using these criteria, in this preliminary study, we have chosen three widely-used and well-tested libraries–LibPNG [9], iGraph [7], and OpenSSL [12]–as our subject libraries. All of these popular programs are included in the OSS-Fuzz project [13] and are frequently subjected to large-scale fuzzing.

To analyze the fuzz blockers in these projects, we conducted analyses based on the results obtained from FuzzIntrospector [5], a recently introduced introspection framework. Despite our limited number of subject programs, this preliminary study has already yielded intriguing insights. For instance, we discovered that *100%* of the top fuzz blockers in LibPNG are *input independent*. This implies that extending the fuzzing time will not necessarily cover these blockers; rather, we may need to develop new fuzz driver(s) or modify the existing one(s) to effectively remove them.

In this study, we aim to answer the following research questions.

- **RQ1.** What types of fuzz blockers have been found in this study?
- **RQ2.** What makes these fuzz blockers challenging for the current fuzzers?

The remainder of this paper is structured as follows. In Section 2, we give an introduction of coverage-guided greybox fuzzing and a motivating example. In Section 3, we describe the design of our study. In Section 4, we share our preliminary results. In Section 5, we share our plan for a full study. We discuss related work in Section 6 before concluding the paper in Section 7.

2 BACKGROUND AND MOTIVATING EXAMPLE

2.1 Background of Coverage-Guided Greybox Fuzzing

Fuzzing is an automated process of repeatedly and intelligently generating "random" inputs (i.e., test cases) and feeding them to the system under test (SUT) to cover more lines of code and discover bugs [36]. Figure 2 shows the common workflow of coverage-guided fuzzing (CGF), which is considered the most scalable and effective fuzzing approach nowadays. Given a program under test (PUT) (e.g., a PDF Reader utility), and a seed corpus of sample program inputs (e.g., PDF files), a CGF fuzzer will (1) select a sample input from the corpus, and then (2) mutate/modify it to generate many new inputs/files, before (3) sending them to the PUT and observing PUT's behaviours. If the newly generated input triggers new PUT's behaviours (e.g., covering a new branch on the control flow graph), the CGF fuzzer will (4) insert that input/file into the seed corpus for further cycles of fuzzing. If some abnormal behaviour is detected—by the crash/bug detection component—the fuzzer will (5) keep the bug-triggering input and prepare a report for further analysis and bug fixing. This loop of five steps will repeat until some specified timeout is reached or the developers/testers decide to stop the fuzzing process. Throughout this process, bugs are detected, and the seed corpus is enlarged to cover more code of the PUT.

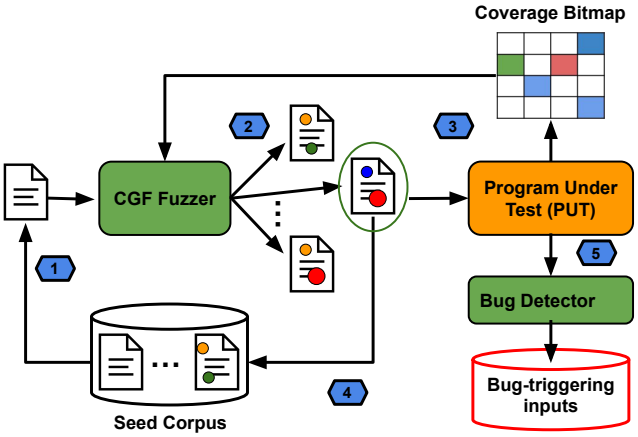

Figure 2: The common 5-step workflow of Coverage Guided Greybox Fuzzing

Note that the program under test in this workflow can be a complete program (e.g., a PDF Reader utility) or a so-called fuzz driver. A fuzz driver is a program which can execute library functions by feeding them with inputs provided by the fuzzer. In practice, fuzz drivers are mainly written by security experts. However, many fuzz drivers (e.g., for the Chromium project [1]) have been written by developers. In the scope of this study, since we are analyzing popular software libraries, fuzz drivers are our main focus.

2.2 Introduction to Fuzz Introspector

FuzzIntrospector [5] is a pretty new yet very effective tool that is designed to help fuzzer developers get a better understanding of their fuzzer's performance and identify any potential blockers. FuzzIntrospector aggregates information like code coverage, hit frequency, entry points etc based on both static analysis passes and dynamic runtime information to give the developer a "birds eye view" of their fuzzer and the in-use fuzz driver(s). Using this toolset, developers have successfully improved coverage achievement and bug found in several case studies such as Xpdf, jsonnet, file, and bzip2 [14].

FuzzIntrospector reports results, including fuzz blockers, for each fuzz driver. It reports the top 12 fuzz blockers based on several metrics such as "non-covered complexity", "unique reachable functions", and "all reachable complexity".

[1] https://github.com/chromium/chromium

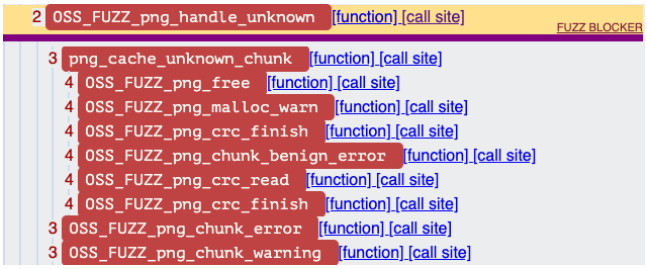

Figure 3: A sample fuzz blocker in LibPNG reported by FuzzIntrospector. The function png_handle_unknown is not reached and so are its callees.

2.3 Motivating Example: A Challenging Fuzz Blocker in LibPNG

Portable Network Graphic (PNG) is a popular image format. It starts with a short header of 8 bytes followed by a series of chunks, each of which conveys certain information about the image. There are mandatory chunks (e.g., IHDR, IDAT, IEND) and optional chunks (e.g., tRNS). LibPNG [9] is the official reference library for PNG images. It supports almost all PNG features and has been extensively tested for over 23 years. In current fuzzing setup for LibPNG, it has only one fuzz driver named *libpng_read_fuzzer*. This fuzz driver follows the instructions in the library's manual [10] to read a given PNG file sequentially using functions such as png_read_info, png_get_IHDR, transformation functions (e.g., png_set_gray_to_rgb, png_set_tRNS_to_alpha), png_read_row, and png_read_end.

FuzzIntrospector ran an analysis on LibPNG using a large seed corpus that was generated after 6+ years of fuzzing LibPNG on OSS-Fuzz [13]. Several CGF fuzzers have been used to fuzz the library such as AFL, Honggfuzz, and libFuzzer. FuzzIntrospector reported 12 fuzz blockers with this fuzz driver and we discuss here an interesting one, as shown in Figure 4.

```
else if ((keep = png_chunk_unknown_handling(png_ptr, chunk_name)) != 0)
{
   png_handle_unknown(png_ptr, info_ptr, length, keep);

   if (chunk_name == png_PLTE)
      png_ptr->mode |= PNG_HAVE_PLTE;

   else if (chunk_name == png_IDAT)
   {
      png_ptr->idat_size = 0; /* It has been consumed */
      break;
   }
}
```

Figure 4: A sample fuzz blocker in LibPNG. The highlighted code has not executed because with the existing fuzz driver, the function png_chunk_unknown_handling returns zero for all the inputs in the corpus.

In this example, the highlighted piece of code has not been executed because the function *png_chunk_unknown_handling* returns zero for all test inputs generated so far. After following our analysis workflow (Section 3), we conclude the root cause is that the fuzz driver being analyzed does not set the list of accepted unknown data chunks and corresponding call-back functions to handle them. To overcome this blocker, only generating more inputs does not help; we need to update the fuzz driver to call the png_set_keep_unknown_chunks function. However, without having a deep understanding of the library, *it is more challenging than it sounds*. We need to add this missing function call with valid arguments (i.e., valid supported unknown chunks and handling functions) to it. It could be even more challenging for an automated fuzz driver generation approach like FUDGE [19] or FuzzGen [32].

Interestingly, we have also noticed that from LibPNG v1.6.0, the library supports a simplified API which hides the details of both LibPNG and the PNG file format itself and if a developer uses this API, the function png_set_keep_unknown_chunks will be automatically invoked with some correct arguments. It means that another option for us to overcome this fuzz blocker is to write a completely new fuzz driver that reads a given PNG image using the simplified API. If we want to do it automatically using FUDGE [19], it may be out of their reach because there is no existing "consumer" code for this API—which is required by the tool—in the LibPNG codebase, to the best of our knowledge. In theory, FuzzGen [32], might help; however, FuzzGen currently only supports some specific types of libraries (e.g., libraries in the Android framework)[2].

We have confirmed our finding by writing a new fuzz driver using the simplified API and the result shows that it successfully uncovered this specific fuzz blocker.

3 STUDY DESIGN

In this section, we discuss the subject selection criteria for our study and a generic workflow that can be applicable to any library.

3.1 Subject Selection

We select software libraries based on their popularity, code size, their diversity in application domains, and the number of existing fuzz drivers. Since we use FuzzIntrospector [5] to identify fuzz blockers, the selected libraries must also be supported by the framework. Moreover, we will only focus on libraries that have low or medium (less than 80%) function reachability and coverage achievement, as reported by FuzzIntrospector. We argue that these are more interesting compared to libraries in which fuzzers have already achieved high reachability and cover most of the codebase.

- **C1-Popularity.** For this, we only consider libraries that have been integrated into the OSS-Fuzz fuzzing platform. It ensures that the libraries are popular and worth analyzing. Moreover, since OSS-Fuzz continuously runs fuzz testing on the libraries, the identified fuzz blockers are more appropriate.
- **C2-Code Size.** Since the analysis requires substantial manual human effort, we focus on medium-size libraries to make the task feasible.
- **C3-Diversity.** The selected libraries should be in different domains. They could share some common properties but all should not be of the same types, e.g., chunk-based file processing libraries (LibPNG, LibJpeg). Moreover, they should

[2] https://github.com/HexHive/FuzzGen/issues/18

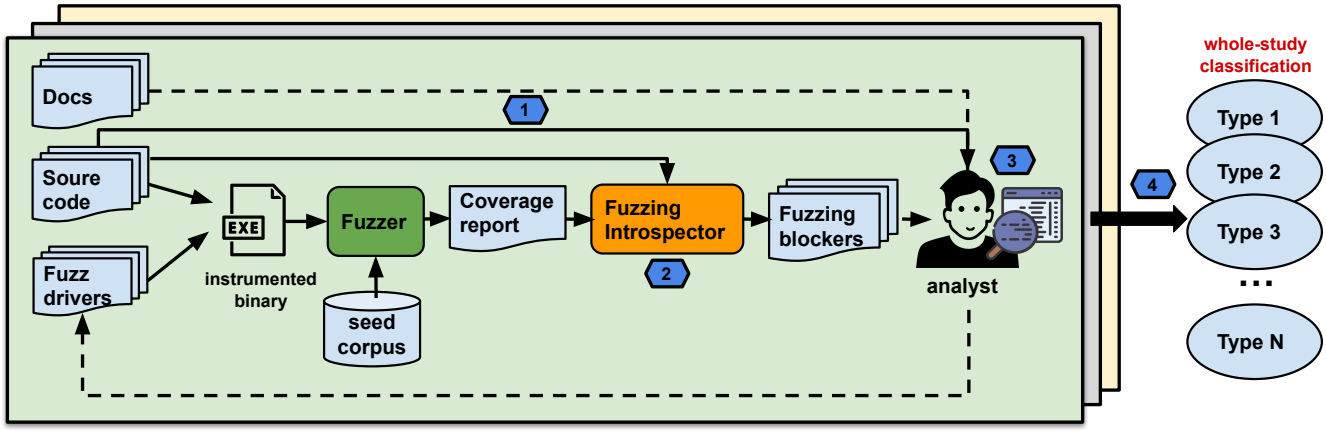

Figure 5: The 4-step workflow to conduct our study. (Step 1 - Manual) Understanding subject program; (Step 2 - Fully Automated with Fuzz Introspector [5]) Identifying fuzz blockers; (Step 3 - Manual) Analyzing fuzz blockers; (Step 4 - Semi-Automated using taint analysis) Classifying fuzz blockers. Dashed lines indicate that the steps/flows are optional.

not only do input parsing; some core algorithms are required (e.g., graph processing and cryptography algorithms).

- **C4-Fuzz Drivers.** We should include both libraries that have only one fuzz driver (e.g., in the case of LibPNG) and libraries that have several fuzz drivers (e.g., iGraph, OpenSSL). This allows us to analyze cases in which developers are aware of the importance of having more fuzz drivers.

Based on these criteria, in this preliminary study, we have selected three popular libraries LibPNG, iGraph, and OpenSSL. These libraries have been fuzzed in several years on the OSS-Fuzz platform with AFL/AFL++, Honggfuzz, and libFuzzer[3].

Table 1 shows the details of these libraries. LibPNG is the reference library handling the PNG file format. iGraph is a collection of network analysis tools with an emphasis on efficiency, portability and ease of use. OpenSSL is a software library for applications that provide secure communications over computer networks against eavesdropping or the need to identify the party at the other end. It is widely used by Internet servers, including the majority of HTTPS websites.

Table 1: Three subject libraries of our preliminary study. We report the code size in Lines of Code (LoC) which is taken from Black Duck Open Hub [2]. Function reachability results are reported by FuzzIntrospector.

Libraries	Size	Functionalities	Function Reachability Result	#Fuzz drivers
LibPNG	105k	Image processing	52%	1
iGraph	520k	Graph analysis	25%	11
OpenSSL	1570k	Cryptography	28%	13

The single fuzz driver in LibPNG tests the core steps to read a PNG image sequentially. The iGraph library has 11 fuzz drivers in total: 8 of them test parsing functions for different graph formats (e.g., UCINET DL [4], edge list, GML [5], GraphML [6], pajek [7]) and 3 of them test graph algorithms such as edge connectivity and vertex separator. The OpenSSL library has 13 fuzz drivers in total: 2 of them test client and server implementations using LibSSL, 3 of them test non-protocol-related components in the LibCrypto (e.g., big number calculations, big number division), and 8 of them test protocol-related components in LibCrypto (e.g., ASN1, ASN1 parsing, x509). This shows the diversity of the fuzz drivers under analysis, satisfying both criteria C3 and C4.

3.2 Analysis Workflow

We design a 4-step workflow to conduct our study as shown in Figure 5. The workflow involves both automated and manual tasks.

3.2.1 ***Step-1. Understanding the subject program.*** It is worth noting that this step is optional. If the analyst is the main developer of the project, it can be skipped. In this step, the analyst reads the available documentations and the structure of the source code to get a high-level understanding of the library under analysis. For instance, in the case study on LibPNG, we relied on the book "PNG: The Definitive Guide" and the library manual [10]. In the case of iGraph and OpenSSL, we focused on their available architectures and manual pages. The analyst should also analyze graphs like function call graphs, and intra-procedural control flow graphs to get a better understanding of the subject program.

In this preliminary study, since function call graphs of these libraries are quite dense, it is hard for us to analyze them. We believe that having a high-level structural representation of the library under analysis would be really helpful. While UML diagrams like class diagram or package diagrams could be helpful, these are rarely available in open-source libraries supported by OSS-Fuzz. We believe that a module dependency graph—in which functions of

[3]For each project, OSS-Fuzz stores a project.yaml file listing project-specific information and active fuzzers (e.g., AFL, libFuzzer.

[4]https://gephi.org/users/supported-graph-formats/ucinet-dl-format/
[5]https://gephi.org/users/supported-graph-formats/gml-format/
[6]http://graphml.graphdrawing.org/
[7]https://gephi.org/users/supported-graph-formats/pajek-net-format/

the same source file should be grouped—seems suitable. However, to the best of our knowledge, there are no such tools available yet. We plan to develop one and share it with the community as a side product of our study.

3.2.2 ***Step-2. Identifying fuzz blockers.*** In this step, we use FuzzIntrospector [5] (Section 2). However, technically any tools or algorithms that are capable of identifying fuzz blockers should fit this workflow. If the analyst does not introduce new fuzz drivers, they can rely on the online FuzzIntrospector's reports for OSS-Fuzz projects, including our three libraries. These reports are periodically generated. If the analyst writes new fuzz driver(s), they would need to re-run FuzzIntrospector to get updated results.

3.2.3 ***Step-3. Analyzing fuzz blockers.*** There could be several fuzz drivers for one library, and currently FuzzIntrospector produces one report for each fuzz driver. Due to that, the analyst should first identify unique fuzz blockers to avoid duplicate works. We have sent a request to the FuzzIntrospector team to generate an aggregated report for all fuzz drivers along with the individual report. It could also reduce the chance of having "false positives". During our analysis of iGraph, we noticed that FuzzIntrospector reported several blockers (e.g., due to the missing attribute tables for edges and vertices) in some fuzz drivers even though these had been covered by other fuzz drivers. If we had an aggregated coverage report for all fuzz drivers, it would not be an issue.

For each unique fuzz blocker, we adopt a step-by-step approach to delve deeper into the source code. Unreachable sections of code can have various levels of depth, so our aim is to progressively investigate further in order to pinpoint the exact cause of the fuzz blocker. To illustrate this, let's consider the code snippet 'if (A != 0)'. Initially, at the surface level, we can determine that A is never equal to 0. However, by delving deeper into the code, specifically to the point where A is defined, we will analyze which function modifies A to make it non-zero. Subsequently, we will examine why this particular function is consistently invoked during the fuzzing process, and so on. Through this iterative process, we will gradually approach the true root cause.

For instance, in the case of the motivating example in Section 2, we first tried to answer the question "Why was the function png_handle_unknown not executed?". Once we know that it was because the function png_chunk_unknow_handling always returns zero in this fuzz driver, we asked ourselves the next question "Why does this function always return zero in this context?" and so on until we knew that the root cause was the function png_set_keep_unknown_function was not included in the fuzz driver.

3.2.4 ***Step-4. Classifying fuzz blockers.*** We aim to classify the fuzz blockers in such a way that we can map them to existing or potential solutions.

To that end, we first divide the fuzz blockers into two groups: input-dependent (a.k.a tainted) blockers and input-independent blockers because the approaches to tackle them are fundamentally different. For input-dependent fuzz blockers (e.g., comparisons with magic numbers, complicated branch conditions) we could improve the core components of the fuzzer itself (e.g., the seed selection algorithm, energy scheduling, mutation operators). However, for input-independent blockers, we must either update the existing fuzz driver or create a new one.

To confirm if a blocker/blocking condition is tainted or not, we use the LLVM Data Flow Sanitizer (DFSan) [4]. We chose DFSan instead of other taint analysis engines because it is LLVM-based and hence could be integrated into FuzzIntrospector more easily.

Once we have classified a blocker as input-dependent or input-independent, we will further classify it based on the actual root cause. For instance, if a blocker is input-independent, the root cause could be that the code is controlled by some missing function arguments, some settings, or the code is executed only if a specific function/list of functions are invoked. We report our initial classification based on the analyses of the three selected projects in Section 4.

It is worth noting that there exist blockers/blocking conditions that are composite conditions in which some predicates are input dependent and some predicates are input independent. To make it less ambiguous, in the scope of this study, we analyze the root cause of the blocker and if the main blocking predicate is input independent, we consider that the blocker is input independent.

4 PRELIMINARY RESULTS

Following the study design and the 4-step workflow presented in Section 3, we have analyzed 12/12 (100%) fuzz blockers in LibPNG[8], 132/132 (100%) fuzz blockers in iGraph[9], and 34/156 (21.8%) fuzz blockers in OpenSSL[10]. After doing deduplication (Step 3), we analyzed 22 unique fuzz blockers in iGraph. It took the first author—who has a Master's degree in Information Technology and had no prior knowledge of the implementation of the selected libraries—three months working part-time (4 hours a day) to complete the analyses with support and guidance from other co-authors.

It is worth noting that we have not completed analyzing all the fuzz blockers in OpenSSL because of the following reasons. First, this is the largest project in our benchmarks with 1.57MLoC. Second, there are some errors in FuzzIntrospector, leading to incorrect or incomplete results. Notably, FuzzIntrospector incorrectly pointed us to the code of OpenSSL v3.0 while we were analyzing fuzz blockers of OpenSSL v1.1.0 and vice versa. Moreover, possibly due to indirect calls and jumps, the static graphs based on which FuzzIntrospector generated the reports are less complete, compared to the other two libraries, making it harder to analyze. We have reported the issues to the FuzzIntrospector team and the team has confirmed some of the issues[11].

Because of these reasons, for this registered report, we decide to use the results from our analyses for LibPNG and iGraph libraries to answer the research questions.

[8]We analyzed the report generated by FuzzIntrospector on 11th Jan 2023. Access link: https://storage.googleapis.com/oss-fuzz-introspector/libpng/inspector-report/20230111/fuzz_report.html

[9]We analyzed the report generated by FuzzIntrospector on 12th Feb 2023. Access link: https://storage.googleapis.com/oss-fuzz-introspector/igraph/inspector-report/20230212/fuzz_report.html

[10]We analyzed the report generated by FuzzIntrospector on 2nd Feb 2023. Access link: https://storage.googleapis.com/oss-fuzz-introspector/openssl/inspector-report/20230202/fuzz_report.html

[11]https://github.com/ossf/fuzz-introspector/issues/967

4.1 RQ1. Types of fuzz blockers

Based on our analysis, 61.7% of fuzz blockers in LibPNG and iGraph are not input dependent (i.e., not tainted). We have used DFSan as a taint analysis tool to confirm all cases in LibPNG but have not done so for iGraph because of some compilation issues.

In the case of LibPNG, we manually annotated the fuzz driver to taint all bytes in the input buffer (using the dfsan_set_label function) and added checks to the blocking predicates (using the dfsan_get_label function). If the label at a predicate is zero, it indicates that the predicate is not tainted. Otherwise, it is tainted. After that, we compiled the annotated fuzz driver and the annotated LibPNG source code before executing the binary with the given seed corpus. Like other taint analysis engines, DFSan could also face the issues of under-tainting and over-tainting, leading to potentially incorrect results. However, we use it in a best-effort manner. Technically, the annotation can be automated since FuzzIntrospector gives us all the details (e.g., line number) of each fuzz blocker. We can then just write an LLVM instrumentation pass to complete the task.

```
// Set several transforms that browsers typically use
png_set_gray_to_rgb(png_handler.png_ptr);
png_set_expand(png_handler.png_ptr);
png_set_packing(png_handler.png_ptr);
png_set_scale_16(png_handler.png_ptr);
png_set_tRNS_to_alpha(png_handler.png_ptr);
```

Figure 6: A cope snippet that enables several transformations in the sole fuzz driver of LibPNG.

4.1.1 Input-independent fuzz blockers (61.7% overall, LibPNG: 12/12, iGraph: 9/22). We further classified these 21 fuzz blockers into different types and sub-types based on their root causes (following the guideline in Step-4 in our presented workflow).

- **Type-1. Fuzz blockers due to wrong function arguments.** We have encountered several instances, particularly within the iGraph library, where the fuzz driver(s) solely incorporate function calls utilizing default or fixed argument values (e.g., NULL, TRUE/FALSE). Consequently, the corresponding sections of code responsible for handling alternative argument values remain unexecuted. As an illustration, the fuzz driver employed in testing the vertex separator algorithm of iGraph exclusively employs a directed graph as an argument, thereby obstructing the execution of functions designed for undirected graphs.
- **Type-2. Fuzz blockers due to missing function call(s).** This type of fuzz blockers refers to the cases in which a function is completely missing in the fuzz driver. We further divide them into four more sub-types.
  - **Type-2.1. Missing calls to overloading functions.** In this sub-type, the fuzz driver under analysis does call a specific version of a function but it does not call its variants or overloading functions (e.g., functions with the same name but having different arguments). For instance, in the fuzz driver that tests the edge connectivity algorithm of iGraph, one function is called in which other calculations inside the maxflow algorithm are disabled. In the same fuzz driver, a specific function is used in which the calculation inside the minimum cut algorithm is not reachable.
  - **Type-2.2. Missing repeated function calls.** For instance, in LibPNG, we have identified a specific fuzz blocker that is only uncovered if the initialization function is called twice. However, it is not the case because the fuzz driver was written just to test the normal image reading procedure.
  - **Type-2.3. Missing function calls to change library settings.** We have noticed several cases, especially in the LibPNG library, in which some transformation code is only executed if some bit has been set in the configuration and the bit is input-independent; it must be set by some function as shown in Figure 6. Suppose we want to test the code for RGB-to-Gray transformation, we should update the fuzz driver[12] to call the png_set_rgb_to_gray function.
  - **Type-2.4. Missing function calls to support more features.** This sub-type is quite similar to Type-2.3. However, the difference is that those missing function calls do not change any settings. Instead, they support new features. For instance, in the motivating example described in Section 2, the fuzz driver misses a function to set up the handling code for some supported unknown chunks.
- **Type-3. Fuzz blockers due to missing different order(s) of function calls.** This type of blocker is interesting. For instance, we have noticed a fuzz blocker in LibPNG that is related to an error-handling piece of code. This can only be executed if the user of the library calls some functions in an unexpected order. Obviously, it is not the case in the existing fuzz driver so the code was not executed, no matter how long we run a fuzzing campaign.
- **Type-4. The blocked code is not reachable.** We have analyzed the fuzz blockers of iGraph and noticed two interesting blockers. They are placeholders for handling errors when using the iGraph's R interface. However, it would never be executed because the R interface sets its own error handlers.

4.1.2 Input-dependent fuzz blockers (9% overall, LibPNG: 0/12, iGraph: 3/22).

- **Type-5: Fuzz blockers due to missing "extreme" inputs.** All three blockers of this type are in iGraph. The library has error handling code for extreme cases such that the number of edges or the number of vertices exceeds some limit.

There are a few blockers in iGraph that could be considered input dependent or input independent. They are blocking the code that handle memory allocation failures which could happen because of some system error[13] or because of the input (e.g., input value leading to large memory requests).

Moreover, we have identified three blockers in iGraph that could be considered false positives because the blocked code was covered by other fuzz drivers. It demonstrates the need of having an aggregated report in FuzzIntrospector.

[12] https://github.com/glennrp/libpng/blob/libpng16/contrib/oss-fuzz/libpng_read_fuzzer.cc

[13] https://stackoverflow.com/questions/18684951/how-and-why-an-allocation-memory-can-fail

It is interesting that no fuzz blockers under our analyses are due to magic number comparisons, which is a well-known type of blocker for fuzzing[14]. It could mean that the current fuzzers can handle them well. However, we also think about other potential reasons. First, FuzzIntrospector only reports the top 12 fuzz blockers so we may see other blockers–some could be relevant to magic numbers or checksums–after uncovering the top ones. Second, due to the current implementation, FuzzIntrospector might miss some blockers. Specifically, to identify fuzz blockers for a specific library L, FuzzIntrospector uses the L's coverage report obtained by running all inputs in its corpus and FuzzIntrospector does not distinguish between the initial seeds (e.g., some sample valid PNG files) and fuzzing-generated inputs. There could exist constraints/predicates that are only satisfied by initial seeds but FuzzIntrospector does not report them as fuzz blockers. We plan to discuss with FuzzIntrospector's team and investigate those cases further in our full study.

4.2 RQ2. What makes them challenging?

Table 2: A mapping from types of fuzz blockers to potential solutions.

Type	Potential Solution	Existing work
Type-1	Automated Fuzz Driver Generation	[19, 32]
Type-2	Automated Fuzz Driver Generation	[19, 32]
Type-3	Automated Fuzz Driver Generation	[19, 32]
Type-4	Unknown	N/A
Type-5	Structure-aware Fuzzing	[11, 15, 44, 47]

In Table 2 we show a mapping from different types of fuzz blockers to potential solutions. We are not aware of any existing solutions for Type-4 blockers and we are not 100% sure if this type of blocker should be addressed because the blocked code is just some placeholder. Regarding other types, in theory, Type-1,2,3 blockers can be addressed by existing automated fuzz driver generation approach and Type-5 blockers can be addressed by structure-aware fuzzers. However, there are several challenges.

First, the research topic of automated fuzz driver generation is under-explored and the existing tools are either closed-source (as in the case of fudge [19]) or do not support all popular libraries out-of-the-box (as in the case of FuzzGen [32]). Moreover, the search space for those tools is huge. The algorithm needs to take into account the validity of functions, their arguments, the order in which those functions should be called, and their dependencies on program states etc.

With regards to Type-5 blockers, tools like AFLSmart [44] and Libprotobuf-mutator [11] could technically be helpful. However, since the limits on the number of edges or vertices can be significantly large, requiring input graphs much larger than usual, current fuzzing algorithms may ignore them in order to maintain efficiency. This issue could be addressed by updating the fitness function and/or the test generation algorithms.

[14]Approaches like RedQueen [17] have produced great results in handling magic numbers and checksums

5 PLAN FOR A COMPLETE STUDY

We have presented the results of our preliminary study mainly on two popular libraries LibPNG and iGraph.

In order to complete the full study and prepare it for a TOSEM journal submission, we intend to expand our research in terms of scope and depth. Our primary objective is to increase the number of subjects included in the study, adhering to our established selection criteria. To make the study feasible, we plan to complete the analyses for OpenSSL and add five more subjects. So the total number of subject libraries in our full study is eight (8). Furthermore, we aim to conduct a more comprehensive investigation for LibPNG particularly. While our current study primarily analyzed 12 top-level fuzz blockers reported by FuzzIntrospector, we recognize the value in conducting a layered analysis that aims to uncover solutions for nearly all encountered blockers. This approach would provide valuable insights into the progressive unblocking of fuzzing blockers, contributing to a more comprehensive understanding of the subject matter.

Reproducibility: To support future research on fuzzing, we will make the artifacts of our study available at *https://github.com/MelbourneFuzzingHub/fuzz_blockers*.

6 RELATED WORK

To the best of our knowledge, once fully completed, this study will represent the first semi-automated and comprehensive examination of fuzzing blockers across multiple well-tested open-source projects. Our research aims to address a significant gap in the existing literature.

Among the closely related works, a notable study was done by Liang et al. [34]. However, it is important to note that their analysis was conducted in a fully manual manner and focused solely on a specific industry library. In contrast, our study endeavors to encompass a broader range of open-source projects, employing semi-automated techniques to explore and understand the various fuzz blockers.

Another line of research that is related includes reflections on fuzzing [21] and review papers [36]. Although these works primarily summarize the state-of-the-art in fuzzing and propose future research directions, they do not delve into analyzing the root causes that impede state-of-the-art fuzzers from surpassing the coverage plateau. In contrast, our study specifically aims to investigate and classify the root causes of fuzz blockers, with the objective of enabling fuzzers to make further progress beyond their current capability.

Studies that analyze and classify bug/vulnerability types [24, 46] also bear relevance to our research. However, it is important to note that their primary objective is to investigate the root causes of bugs and vulnerabilities once they have been identified. Essentially, these studies focus on the "knowns". In contrast, our study is centered around exploring the root causes of fuzz blockers, which hinder the progress of fuzzers. By doing so, we aim to shed light on the "unknowns" factors that impede the efficacy of fuzzers and subsequently enable further advancements in the field.

The fuzzing research community has proposed several novel ideas to improve the effectiveness and efficiency of fuzzing in the

past few years [15, 22, 23, 27, 43, 44, 47]. While they focus on improving and addressing some specific challenges, we aim to build a high-level view of all the fuzz blockers.

7 CONCLUSION

In this registered report, we have emphasized the significance of conducting a comprehensive study to investigate the underlying factors contributing to the lack of progress exhibited by state-of-the-art fuzzers after reaching their plateau. Such an investigation holds the potential to illuminate new research avenues in the field of fuzzing, providing valuable insights and directing researchers towards under-explored areas. Our preliminary findings, derived from an examination of three popular libraries (LibPNG, iGraph, and OpenSSL), indicate that a significant number of fuzz blockers are not inherently tied to input dependencies. Rather, these blockers can be effectively addressed through the generation or modification of fuzz drivers. However, it is worth noting that this specific research area remains relatively unexplored. Through our detailed classification of fuzz blockers, we aim to offer guidance to researchers working in this domain, highlighting specific root causes that can be leveraged to enhance techniques. Additionally, we outline our plan to complete this study and submit it to a journal.

During the course of our study, we have identified a requirement for the development of supportive tools aimed at enhancing code comprehension. For instance, there is a need for tools facilitating the construction of module dependency graphs, which can aid in understanding the relationships between different code modules. Additionally, we have observed the presence of incomplete call graphs, resulting in incomplete outcomes for FuzzIntrospector.

8 ACKNOWLEDGEMENT

We thank Navid Emamdoost, David Korczynski, Jonathan Metzman, and Addison Crump for sharing their great insights into FuzzIntrospector, OSS-Fuzz, FuzzBench, and DFSan. We also thank the anonymous reviewers for their constructive and encouraging feedback. This research received funding from Google and it was also partially funded by the Australian Government through an Australian Research Council (ARC) Discovery Early Career Researcher Award (DE230100473).

REFERENCES

[1] [n. d.]. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/
[2] [n. d.]. Black Duck Open Hub. https://www.openhub.net/
[3] [n. d.]. ClusterFuzz. https://google.github.io/clusterfuzz/
[4] [n. d.]. Data Flow Sanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html
[5] [n. d.]. Fuzz Introspector. https://github.com/ossf/fuzz-introspector
[6] [n. d.]. Honggfuzz. https://github.com/google/honggfuzz
[7] [n. d.]. Igraph The Network Analysis Package. https://igraph.org/
[8] [n. d.]. libFuzzer – a library for coverage-guided fuzz testing. https://github.com/llvm-mirror/llvm/blob/master/docs/LibFuzzer.rst
[9] [n. d.]. libpng home page. http://www.libpng.org/pub/png/libpng.html
[10] [n. d.]. libpng manual. http://www.libpng.org/pub/png/libpng-manual.txt
[11] [n. d.]. libprotobuf-mutator. https://github.com/google/libprotobuf-mutator/
[12] [n. d.]. OpenSSL crytography and SSL/TLS toolket. https://www.openssl.org/
[13] [n. d.]. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz
[14] [n. d.]. Successful case studies of FuzzIntrospector. https://github.com/ossf/fuzz-introspector/blob/main/doc/CaseStudies.md
[15] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *NDSS*.
[16] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1597–1612.
[17] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.
[18] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3255–3272.
[19] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 975–985.
[20] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing.. In *USENIX Security Symposium*, Vol. 19.
[21] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Softw.* 38, 3 (2021), 79–86.
[22] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.
[23] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* 45 (2016), 489–506.
[24] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software* 152 (2019), 165–181.
[25] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 736–747.
[26] Karine Even-Mendoza, Cristian Cadar, and Alastair F Donaldson. 2022. CsmithEdge: more effective compiler testing by handling undefined behaviour less conservatively. *Empirical Software Engineering* 27, 6 (2022), 129.
[27] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2829–2846. https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi
[28] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.
[29] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing.. In *USENIX Security Symposium*. 2577–2594.
[30] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 230–243.
[31] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*. 45–48.
[32] Kyriakos K Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. Fuzzgen: Automatic fuzzer generation. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 2271–2287.
[33] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 809–814.
[34] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. 2018. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 562–566.
[35] Dongge Liu, Jonathan Metzman, Marcel Böhme, Oliver Chang, and Abhishek Arya. 2023. SBFT Tool Competition 2023–Fuzzing Track. *arXiv preprint arXiv:2304.10070* (2023).
[36] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.
[37] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 1393–1403.

[38] Sebastian Österlund, Elia Geretto, Andrea Jemmett, Emre Güler, Philipp Görz, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. 2021. Collabfuzz: A framework for collaborative fuzzing. In *Proceedings of the 14th European Workshop on Systems Security*. 1–7.

[39] Lianglu Pan, Shaanan Cohney, Toby Murray, and Van-Thuan Pham. 2023. Detecting Excessive Data Exposures in Web Server Responses with Metamorphic Fuzzing. *arXiv preprint arXiv:2301.09258* (2023).

[40] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative type-aware mutation for testing SMT solvers. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–19.

[41] Hui Peng and Mathias Payer. 2020. Usbfuzz: A framework for fuzzing USB drivers by device emulation. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 2559–2575.

[42] Van-Thuan Pham, Manh-Dung Nguyen, Quang-Trung Ta, Toby Murray, and Benjamin I.P. Rubinstein. 2021. Towards Systematic and Dynamic Task Allocation for Collaborative Parallel Fuzzing. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering : NIER Track*.

[43] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.

[44] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1980–1997.

[45] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis.. In *OSDI*, Vol. 20. 667–682.

[46] Yang Song and Oscar Chaparro. 2020. BEE: a tool for structuring and analyzing bug reports. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1551–1555.

[47] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.

[48] Qian Zhang, Jiyuan Wang, and Miryung Kim. 2021. Heterofuzz: Fuzz testing to detect platform dependent divergence for heterogeneous applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 242–254.

[49] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. 2023. Fuzzing Configurations of Program Options. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–21.

[50] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970.