THE UNIVERSITY
of ADELAIDE

Fuzz Driver Generation

Supun Jeevaka Dissanayake

A thesis submitted for the degree of
MASTER OF PHILOSOPHY
The University of Adelaide

June 27, 2022

Contents

Abstract xi

Declaration of Authorship xiii

Acknowledgements xv

1 Introduction 1
1.1 What is fuzzing? 1
1.1.1 Types of fuzzing 2
1.1.2 Fuzzers 2
1.1.3 Fuzzer performance metrics 3
1.2 What is a fuzz driver? 3
1.3 The relationship between the code complexity and the fuzzing campaign 4
1.4 Research objectives and questions 5
1.5 Research methodology 7
1.6 Thesis contributions 8
1.7 Other contributions 8

2 Fuzz driver generation for fuzz testing 11
2.1 Introduction 11
2.2 Background on fuzz drivers 14
2.3 Methodology 15
2.4 Findings 18
2.4.1 Context of fuzz driver mentions in fuzzing research 18
2.4.2 Fuzz driver requirements 19
2.4.3 Fuzz driver development practices 19
2.4.4 Fuzzer and fuzz driver evaluation techniques 21
2.5 Context of fuzz driver mentions in fuzzing research 21
2.6 Requirements for fuzz driver development 22
2.6.1 Characteristics of a fuzz driver 22
2.6.2 Function identification 23
2.6.3 Path identification 23
Reaching hard to reach components of the code 23
2.6.4 Program state and control flow identification 24
2.6.5 Types of bugs 24
2.6.6 Garbage collection in fuzzing 24
2.6.7 Importance of the seed corpus 25
2.6.8 Accommodating parallel fuzzing and multi-threading 25
2.6.9 Speed and timeouts 25
2.6.10 Difficulties of manual fuzz driver development and the need for automation 25
2.7 Fuzz driver development 26
2.7.1 Fuzz driver generation steps 26

2.7.2 Source code analysis for target selection 26
Human interference: target identification by a software developer 27
Annotations/code comments 27
Unit tests 28
Abstract-syntax tree 29
Program slicing 29
Function signature monitoring 30
2.7.3 Function dependency identification and fuzz driver synthesis . . 31
Human interference: function dependency identification by a software tester 32
Program analysis 32
Avoidance of function dependency identification 34
2.7.4 Interfaces fuzzed 35
2.7.5 The use of fuzzers 36
2.7.6 Availability of source code for research replication 36
2.8 Evaluation methods 37
2.9 Fuzz driver development software that does not associate with a research study 38
2.10 Research gaps and future directions 39
2.11 Limitations and threats to validity 39
2.12 Summary 40

3 What is the best fuzz driver generation strategy? 43
3.1 Introduction 43
3.2 Background and related work 44
3.2.1 Fuzz Target Generator (FTG) 45
3.2.2 FuzzGen 48
3.2.3 Other fuzz driver generators 51
3.2.4 Fuzzer metrics for LibFuzzer 52
3.2.5 Code complexity analysis 53
Cyclomatic Complexity 53
Halstead Metrics 54
3.3 Methodology 55
3.3.1 Collection of manually developed fuzz drivers 55
3.3.2 Generating fuzz drivers 56
3.4 Findings 57
3.4.1 Fuzzing 57
Fuzz driver compatibility 58
3.4.2 Analysis of bug identification 58
3.4.3 Analysis of code coverage 60
3.4.4 Analysis of code complexity measures 61
Analysis of cyclomatic complexity 62
Analysis of Halstead metrics 64
3.5 Limitations and threats to validity 67
3.6 Summary 67

4 Conclusion and future work 71
4.1 Conclusion 71
4.2 Future directions 72
4.2.1 Novel ways to improve information transfer from function signatures to a fuzz driver 72

4.2.2 Novel ways to automate unit test to fuzz driver conversion . . . 72
4.2.3 Minimising human effort in semi-automated fuzz driver development . . . 72
4.2.4 Inclusion of single function fuzz driver generation capacity for state of the art automatic fuzz driver generators . . . 73
4.2.5 Development of fuzz drivers for multiple libraries . . . 73
4.2.6 Comparison of micro fuzzing and fuzzing campaigns with fuzz drivers . . . 73
4.2.7 Fuzz driver development automation for multiple programming languages . . . 73
4.2.8 More applications of function importance ranking algorithms . 74

5 Appendix A: Selected studies in the systematic literature review 75

List of Figures

2.1 Fuzz testing life-cycle. 12
2.2 Number of papers that mentions fuzz drivers vs the publication year. . 12
2.3 Distribution of papers along publication years. 13
2.4 Overview of the methodology. 15
2.5 Stages of fuzz driver development process. 27

3.1 Human involvement in semi-automated fuzz driver development. . . . 45
3.2 Fuzz driver development through FTG. 46
3.3 Fuzz driver development through FuzzGen. 50
3.4 Systematic collection of manually developed fuzz drivers. 56
3.5 Three sets of fuzz drivers from gathered target functions. 57
3.6 The number of target functions that are compatible with all three fuzz driver development techniques. 61
3.7 Cyclomatic complexity vs code coverage of manually written fuzz drivers. 62
3.8 Cyclomatic complexity vs code coverage of FTG fuzz drivers. 63
3.9 Cyclomatic complexity vs code coverage of FuzzGen fuzz drivers. . . . 63
3.10 Halstead volume vs code coverage of manually written fuzz drivers. . . 65
3.11 Halstead volume vs code coverage of FTG fuzz drivers. 66
3.12 Halstead volume vs code coverage of FuzzGen fuzz drivers. 66

List of Tables

2.1 Search terms and number of Google Scholar results. 17
2.2 Prioritisation of fuzz driver generation in research papers. 21
2.3 Fuzz driver development strategies. 26
2.4 Types of fuzzers. 36
2.5 Open-Source access to findings. 36
2.6 Fuzzer evaluation methods. 37
2.7 Fuzz driver evaluation methods. 38

3.1 Fuzz driver compatibility with target functions. 58
3.2 Number of crashes, false positives and bugs. 59
3.3 Percentage of bugs to projects. 59
3.4 Code coverage. 60
3.5 Percentage increase in average code coverage when cyclomatic complexity (CC) of the target function goes beyond 50. 64

A Selected studies in the systematic literature review. 75

University of Adelaide

Abstract

Fuzz Driver Generation

by SUPUN JEEVAKA DISSANAYAKE

Poor software quality has led to tremendous costs and safety disasters, thus, software defects make the news with alarming regularity. Fuzzing is a bug detection technique. In particular, it is a software testing method where a stream of random input is sent to an application to stress the application and cause unexpected behaviour, resource leaks or crashes. When it comes to fuzzing software libraries, a fuzz driver plays an important role because it is the binder between the fuzzer and the target program. Traditionally fuzzing was used in closed-source platforms and also it is used to find vulnerabilities in kernels. However, recent developments show that fuzzing is now applied to open-source libraries.

This research study analyses the role of a fuzz driver in the domain of fuzzing to recognise its importance, applications, techniques, challenges and future directions. This study intends to explore the state-of-the-art fuzz driver development strategies and identify trends in research and areas of potential improvements. We identified that fuzz driver generation is mainly seen as a minor activity in fuzzing research. It was evident that the development of a fuzz driver is laborious and time-consuming in nature but multiple innovative methodologies have been adopted in recent years to ease this task

There are three main techniques to develop a fuzz driver: software developers manually writing a fuzz driver, semi-automatic generation of a fuzz driver through human-in-the-loop approaches and fully automatic generation of a fuzz driver. This research study evaluates these techniques through case studies and empirical analysis to recognise the best state-of-the-art fuzz driver generation strategy available for researchers and software testers. Our results show that manually developed fuzz drivers still outperform other methodologies in terms of performance but our results show how other methodologies could surpass their performance levels. Furthermore, this study analyses the effect of varying complexity levels of target functions on the performance of the fuzzing campaigns initiated through multiple fuzz driver generation techniques.

Declaration of Authorship

I, Supun Dissanayake certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name, in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission in my name, for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Adelaide and where applicable, any partner institution responsible for the joint-award of this degree.

I acknowledge that copyright of published works contained within this thesis resides with the copyright holder(s) of those works.

I also give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library Search and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

I acknowledge the support I have received for my research through the provision of an University of Adelaide School of Computer Science Scholarship.

Supun Jeevaka Dissanayake

December 2021

Acknowledgements

This study was made possible by the cooperation and assistance received from a number of people during my study at the University of Adelaide. My academic journey at the University of Adelaide would not have been possible without my three supervisors, Associate Professor Markus Wager, Dr Yuval Yarom and Dr Christoph Treude. It is through their wisdom and encouragement that this work has come to fruition.

I would like to thank Associate Professor Markus Wagner for his useful comments and guidance provided throughout my time at the University of Adelaide, and for his valuable support, encouragement and strength during my studies.

I would like to thank Dr Yuval Yarom for constantly helping me through difficult technical tasks that I carried out during my time at the University of Adelaide. It was a great honour to work with such an inspiring and outstanding Cyber Security research group headed by Yuval at the School of Computer Science. Furthermore, I am grateful that my scholarship was covered through Yuval's startup funding grant.

I would like to thank Dr Christoph Treude for the guidance, comments and kind help provided during my research journey. I am most grateful to him for inviting me to join him at the University of Adelaide. Also, his great support allowed me to carry out collaborative studies with the Federal University of Pernambuco and publish a very high-quality research paper.

My work was supported by the University of Adelaide School of Computer Science scholarship, which I gratefully acknowledge. This Scholarship allowed me to focus on my studies by covering all of my living expenses.

I also would like to thank my parents and my close friends in Adelaide for all their kind support, love and care provided throughout my research journey.

Chapter 1

Introduction

Software Quality Assurance is one of the most significant aspects of the software development process. There are a variety of quality assurance techniques adopted in the industry to nullify modern security threats faced by software systems. Fuzzing is regarded as one of the most effective software vulnerability identification techniques to identify security and reliability issues in computer software.

Fuzzing was invented by Barton Miller at the University of Wisconsin [106]. Fuzzing is a technique for identifying software faults. The core concept behind fuzzing is to send randomly generated input to a target program and check whether the program behaves correctly. If the program crashes, the fuzzer knows that something is wrong and it can provide the input that causes the problem and the location where the problem occurred to the software tester for further analysis.

Fuzzing is widely used to identify software exploits in penetration testing. Software testers and researchers use fuzzing as a technique to identify software vulnerabilities before those vulnerabilities are identified by attackers. Therefore, defensive measures can be taken to protect the software product. Software giants such as Adobe [1], Cisco [26], Google [51], and Microsoft [104] use fuzzing as part of their secure development practices. It is evident from recent research studies that both open-source software developers [131] and security auditors [181] have adopted fuzzing to enhance the security in their software and prevent vulnerabilities.

There are a variety of software projects that involve fuzzing that is available as open-source products. For example, GitHub [45] itself contains thousands of projects with fuzzing entities. Furthermore, fuzzing research is a common occurrence in software engineering and security conferences. However, there are multiple areas of fuzzing that are yet to be explored, hence the motivation behind this research thesis.

1.1 What is fuzzing?

The term "fuzz" comes from the work of Miller et al. [106], published in 1990. Following that research study, fuzzing has appeared in a variety of research scenarios, including in dynamic symbolic execution [150], complexity testing [174], kernel testing [77], GUI testing [152], grammar-based test case generation [73] etc.

There is a range of terminology in fuzzing. The terms "Fuzzing" or "Fuzz Testing" refers to the process of running the target program with a set of input that it might not be expecting to receive. This process triggers behaviour that was not desired by the developer of the target program. "Fuzzer" is the software that carries out fuzzing. A "Fuzzing Campaign" is the execution of the fuzzer on the target function with a specific correctness policy. The purpose of the fuzzing campaign is to identify bugs that violate the correctness policy. Also, the terms "Seed Input" or "Seed Corpus" refer to the initial set of inputs that serves at the start point of the fuzzing campaign.

1.1.1 Types of fuzzing

Fuzzing can be divided into black box fuzzing, white box fuzzing and grey box fuzzing. Black box fuzzing does not consider any knowledge regarding the target program when it generates input. Black box fuzzing further subdivides into generational and mutational black box fuzzing. Inputs are generated from scratch for generational black box fuzzing and on the other hand, mutational black box fuzzing takes one or more seed inputs and modifies them to generate further inputs for the fuzzing campaign. These modifications include processes such as flipping random bits in the seed input. Peach fuzzer is a well known black box fuzzer [17].

White box fuzzing is based on a methodology called symbolic execution [18]. White box fuzzing uses program analysis to systematically reach deeper parts of the program by increasing the code coverage. On this occasion, the code coverage is the depth of the source code covered by the fuzzer in its fuzzing campaign. White box fuzzing practices have thorough knowledge regarding the target program that is under test.

Grey box fuzzing uses the instrumentation of the program to receive lightweight feedback about the program and then uses that information for the fuzzing campaign [17]. Grey box fuzzers will instrument all the branches in the target program and send the initial seed corpus to the compiled target program. Then the fuzzer mutates these seed inputs to generate new inputs. These generated inputs go on to cover new locations of the code by increasing the code coverage. The fuzzer receives code coverage feedback and then using this information the grey box fuzzer tries to reach deeper parts of the code. Some grey box fuzzers use memory-related bug detection platforms which are called sanitizers that provide information regarding the bug. LibFuzzer [94] is an example of a grey box fuzzer with AddressSanitizer [138] acting as its sanitizer. Another name for grey box fuzzing is coverage-guided fuzzing. In this study, we focus on grey box fuzzing.

1.1.2 Fuzzers

Fuzzer is the software that triggers fuzzing; fuzzers enable the fuzzing process for software testers. LibFuzzer [94] and American Fuzzy Lop (AFL) [179] are examples of fuzzers that are widely used by software testers. Both fuzzers give feedback to their users in terms of code coverage, the number of bugs, executions per second etc.

LibFuzzer [94] is produced by Google and it is aligned with fuzz benchmarks of FuzzBench [44] and OSS-Fuzz [140]. FuzzBench is a standard set by Google to evaluate fuzzer performance; OSS-Fuzz is an open-source fuzzing platform from Google that allows vulnerability detection in security-critical software. LibFuzzer fuzzes the program till it crashes. A system crash is when the software system or an operating system no longer works by stopping all of its functions. The fuzzing campaign of LibFuzzer aborts when it finds a crash or when the fuzzer times out. The time out can also be manually set by a software tester for the target program.

AFL [178] also allows software testers to carry out fuzzing campaigns for their desired software library. Similar to LibFuzzer, AFL is a coverage-guided fuzzer; it tries to maximise the code coverage to improve its fuzzing capacity. If it identifies an input that exercises a new branch, this new input is added to the seed corpus to improve the code coverage in its next cycle.

The advances in coverage guided fuzzing (i.e. grey box fuzzing) [99, 138, 178] allowed the production of software tools with the ability to reach deeper sections of the program code to identify significant bugs. Due to this reason, fuzzing is widely adopted in commercial and open-source software. Google adopted multiple fuzzing platforms to test security in C and C++ libraries in both their internal and external code bases [7,

62]. Google offers OSS-Fuzz through its ClusterFuzz project and they managed to find above 10000 bugs in over 200 open-source projects through fuzzing [140].

Programming languages C and C++ seem to be the most popular for fuzzing related activities. This is mainly due to these programming languages containing unsafe characteristics in memory management, which makes them prone to vulnerabilities [7]. AddressSanitizer helps the identification of bugs by instrumenting source code with multiple checks, which identifies undefined behaviour in the C/C++ code regarding memory corruption [138].

1.1.3 Fuzzer performance metrics

The purpose of fuzzing is to identify vulnerabilities in the program code. Therefore, the metric that indicates the vulnerability of a program code is the number of bugs identified through fuzzing. A software bug is a flaw, error or fault in a computer program that results in producing an unexpected/incorrect result or behaving in an unintended manner compared to its intended action [157]. Some of the example bugs identified through fuzzing are buffer-overflows, integer overflows, stack overflows, uninitialised memory etc., which cause security vulnerabilities. Research studies that focus on applying fuzzing for their program code, conclude on the effectiveness of the fuzzing campaign based on the number of bugs identified [43, 129, 134, 164].

The code coverage is the degree to which the target program is executed during the fuzzing campaign. One of the measures of code coverage is block coverage, which is the covering of straight-line code sequences without any branches. Code coverage is another measurement that identifies the effectiveness of fuzzing. In particular, LibFuzzer [94] measures the block coverage of the target program, which is one of the measurements that we considered in this thesis.

Similarly, other than identifying bugs and code coverage, there are other factors to measure the effectiveness of the fuzzing campaign [40, 43, 129, 147]. These include fuzzing time [83, 113], fuzzing speed [42], number of inputs per crash [162] etc.

1.2 What is a fuzz driver?

One of the steps in the fuzzing life cycle is the writing of a fuzz driver, which is a testing harness designed to exercise the software library code by passing inputs to the target program. The fuzz driver acts as the bridge between the fuzzer and the target program by directing the input from the fuzzer into the target program. Therefore, it plays an essential role in the quality of the fuzzing campaign. If the fuzz driver manages to capture the required aspects of the target program for fuzzing, the fuzzing campaign will have high code coverage and a high rate of bug identification and vice versa.

Some research studies propose to keep the fuzz driver small and only target a single function at a time [28, 73, 94]. On the other hand, some other studies promote fuzzing the whole code base [7, 62, 183]. Writing a fuzz driver for a small project does not take much effort since there is a low number of functions; only a few fuzz drivers should be written by software testers. However, when the size of the project increases to a large codebase, there are hundreds of locations in the code that could be benefited from a fuzzing campaign. Some fuzz drivers are compatible with multiple fuzzers such as LibFuzzer [94], AFL [178], HongFuzz [59] to run a fuzz test.

To develop a good fuzz driver, the software tester must have a good understanding of the target code, fuzz driver writing methodology and the fuzzer. The research study of Kelly et al. [73] claims that the organisation should aid software testers in teaching

to write accurate fuzz drivers for their existing code base. Writing an efficient fuzz driver is a time consuming task that requires deep knowledge regarding the target codebase according to multiple research studies [7, 23, 50, 62, 65, 73, 90, 91, 114, 135, 142, 147, 149, 182, 186]. The software tester should clearly understand the details about the interface of the target program and how it functions. For example, let's say that the target function is an image format library. Then the software tester should know the image content specifics, format parsing operations, configuration of library calls etc. [7]. Furthermore, crashes such as function pre-condition violations due to the lack of understanding of the target function are not helpful for the fuzzing campaign [7]. The deterministic behaviour of a fuzz driver is important since once a bug is identified, the fuzzer should have the ability to reproduce that bug for further analysis with the exact malformed inputs that caused the crash in the first place.

There are multiple ways of developing a fuzz driver. On most occasions, software testers write the fuzz driver manually by themselves to test the program code. However, during recent times, many research studies try to automate this manual task [7, 62, 73, 183]. Attempts were taken to generate fuzz drivers semi-automatically with human-in-the-loop approaches. For example, the work done by Kelly et al. [73] requires the software tester to comment on the function that requires fuzzing and then generates a fuzz driver for that commented function. On other occasions, projects like Fudge [7] scan the code, identifies functions, function dependencies, data flow etc. and synthesise a fuzz driver automatically.

When fuzz drivers are developed through the above-mentioned methods, the identification of bugs and code coverage seems to indicate their effectiveness as mentioned in Section 1.1.3. Results of research studies [7, 62, 73, 183] show that the number of bugs and code coverage are the key metrics that show the effectiveness of the fuzz drivers. On occasions, two of the fuzz driving methods have been compared with each other on a static level [183]. However, no study shows the effectiveness of the three different fuzz driver development strategies from multiple states of the art tools compared against each other for the same code base, in identical environments with dynamic analysis of their performance. Thus opening up a research area to be explored.

1.3 The relationship between the code complexity and the fuzzing campaign

When it comes to identifying the effectiveness of the fuzz driver and the fuzzer performance, research studies tend to only focus upon metrics of the fuzzer such as the number of bugs, code coverage, and executions per second to explore the fuzzer effectiveness. Therefore, there is a requirement to analyse the effectiveness of external factors such as the quality of the target program in terms of code complexity for the performance of the fuzzing campaign. There is no research study carried out to compare the fuzz driver performance along with the code complexities of the target code. Therefore, we aim to explore whether external factors such as the quality of the target code have any effects on fuzzing and whether multiple fuzz driver development strategies have any effect on these external factors and vice versa.

One of the most widely discussed and explored research areas in software quality assurance is code complexity analysis. Source code complexity analysis also appears in fuzzing research. The research study from Shudrak and Zolotarev [146] propose the use of complexity metrics to improve fuzzing. They use complexity assessment metrics to analyse the binary code and come up with solutions to decrease the time

cost of the fuzzing campaign. Code complexity measurement techniques such as cyclomatic complexity [100] measure the number of linearly independent paths through the program to measure the stability and the program confidence. Similarly, Halstead metrics [54] measure the programs module complexity by calculating the number of operations and operands. The study from Shudrak and Zolotarev [146] claims that Halstead metrics shows the best performance in terms of identifying vulnerable routes.

The work from Li et al. [88] proposes the use of static analysis to evaluate the complexity of the function. Then they use function complexities to aid input seed prioritisation when fuzzing a software program. On this occasion, they use cyclomatic complexity [100] and Halstead metrics [54] to identify function code complexity. They claim that depending on the code complexity, the seed input should be carefully selected to improve the code coverage of the fuzzing campaign.

Therefore, it is evident that cyclomatic complexity [100] and Halstead metrics [54] appear multiple times in fuzzing related studies. Furthermore, these two metrics are used in other software testing methods such as unit testing to identify the complexity and the maintainability of the source code before the code is tested with unit tests [3, 37, 64, 123].

1.4 Research objectives and questions

Fuzz drivers are an essential aspect of the fuzzing life-cycle. There are a variety of reasoning behind fuzz driver development and similarly, there are a variety of methodologies adopted by researchers and software testers to develop fuzz drivers. It is evident [7, 62, 70, 91] that fuzz driver development largely requires deep knowledge of the code and a manual effort to write fuzz drivers. This brings up multiple challenges for software companies.

Since the introduction of open-source software such as AFL [178] and LibFuzzer [94], there are standardised methods for fuzz driver development. These fuzzers are widely used in the software testing domain hence multiple studies are carried out around these fuzzers and fuzz drivers. Furthermore, with the rise of open-source fuzzers, research studies start to explore the automation of fuzz drivers which is rather a manual task [62].

As mentioned in Section 1.2, fuzz drivers are written manually, generated semi-automatically with human-in-the-loop approaches and generated fully automatically. However, most of the semi-automatic and automatic fuzz driver development tools are not available for industry usage nor available as open-source products with only a few exceptions [7, 73]. Hence the replication of results in most of these studies is a tall order.

Research studies that develop semi-automatic and automatic approaches show their fuzzing prowess by comparing them against the manually developed fuzz drivers. However, they all carry out these experiments in selected data sets to show their effects, hence there is a lack of large scale studies with replicable results to implicate their true effectiveness in an ideal scenario.

No study compares all automated fuzz driver development strategies from multiple open-source fuzz driver development products to show their effectiveness against manually developed fuzz drivers. Furthermore, all the studies that compare manually developed fuzz drivers against automated methods contain human bias since manually developed fuzz drivers are developed by a single software tester for comparisons. Therefore, depending on the domain knowledge and the ability of the programmer, the results could vary, thus adding bias to the research outcomes. Hence, a research study

is required to compare these three fuzz driver development techniques by neglecting human bias.

Code complexity analysis has been widely used to improve different aspects of the fuzzing campaign [88, 146, 168]. As we identified in Section 1.3, code complexity analysis appears as a sub-process when trying to improve path coverage by improving the effectiveness of the input. Moreover, some studies focus on using complexity metrics to identify the maintainability and the testability of the source code. However, there is no study carried out to analyse code complexity metrics to aid or improve fuzzing campaigns by identifying the effectiveness of fuzz drivers. A deep research study is required to identify the effect of the complexity of the target code has on fuzzing campaigns initiated by different types of fuzz drivers mentioned in Section 1.2. Therefore, we formulate the following three research questions:

- RQ1: What are the fuzz driver generation practices adapted by researchers?
    - Software testers follow multiple techniques and methodologies to develop a fuzz driver. There are multiple requirements around fuzz driver development. There are multiple applicability criteria and challenges around fuzz driver development. There are guidelines that a tester should strictly follow to develop an effective fuzz driver. Multiple studies explore the literature around fuzzing [89, 99, 102], however, such study is not dedicated to fuzz drivers to explore all the themes around fuzz driver development to understand the domain in a more clear light. Therefore, we aim to explore the art of fuzz driver development and its practices. We aim to observe the main objectives, challenges, common practices and strategies of developing fuzz drivers in fuzzing campaigns. We aim to identify criteria and guidelines used in the industry to assess and evaluate the quality of a fuzz driver and the availability of open-source tools to ease, enhance and generate fuzz drivers.
- RQ2: How fuzz drivers generated through different strategies affect bug identification and code coverage?
    - We aim to analyse the best performing fuzz driver development practices that are available for a software tester. When it comes to fuzz driver development, we identified that a fuzz driver could be developed manually by a software tester, they could be developed via a semi-automatic methodology with a human-in-the-loop approach and they could be generated automatically through a fuzz driver generation software. We aim to compare these three strategies and observe the best methodology for generating a fuzz driver for the fuzzing campaign. We aim to explore and analyse how different fuzz driver development methods affect the bug identification and code coverage of the target program and propose the best fuzz driver development technique for software testers with potential enhancement strategies.
- RQ3: Does the code complexity of the target code have any effect on the fuzz driver performance?
    - We aim to identify whether the code complexity of the target code has any effect on different types of fuzz drivers and their fuzzing campaign performance. We aim to analyse how these target code complexities would affect the development of manually developed fuzz drivers, semi-automatically

developed fuzz drivers with human-in-the-loop approaches and fully automatically developed fuzz drivers. Then using the results, we aim to identify the effectiveness of the above-mentioned fuzz driver development techniques when the code complexity increases and we propose strategies to improve their current features to enhance their abilities to fuzz complex code.

1.5 Research methodology

This thesis explores and addresses different aspects of fuzz driver generation methodologies in the domain of fuzzing. Each chapter in this thesis has its contribution while fulfilling the overall goal of addressing to improve fuzz driver development practices. Therefore, each chapter contributes to different areas of research improvement. We answer our three research questions in each of the chapters.

Chapter 2 follows a well-known systematic literature review (SLR) approach [80]. Subsequently, Chapter 3 contains a case study and an empirical study to investigate key aspects related to fuzz drivers and to propose, implement and theorise new methodologies and research directions.

We used multiple methodologies and technologies to answer these three research questions. We used Google Scholar [136] as the search engine to gather research studies for the SLR and Zotero [190] as the technology to extract and manage papers. We carried out subsequent duplication removals, category identifications, simplification of search structures and theme analysis for the SLR to come out with actionable outputs for the fuzz driver development domain.

For all the fuzzing related experiments, we used LLVM LibFuzzer [94]. LibFuzzer allowed us to fuzz software libraries for a given period to identify required metrics such as the number of bugs, executions per second and code coverage for our experiments.

For code complexity analysis we used code complexity identification techniques such as cyclomatic complexity [100] and Halstead metrics [54]. We used an open-source code repository, GitHub [45] as the source to gather all the open-source code to carry out our experimental studies in Chapters 3.

Chapters 2 and 3 of this thesis derive from each of the research studies that we carried out. These research studies are carried out in collaboration with my supervisors. I carried out the main components of this research study; this thesis uses the standard pronoun "We" to report collaborative research in the domains of software engineering, computer science and security. The chapters in this thesis are arranged in the following manner.

- Chapter 1 describes the background of fuzzing, fuzzers, fuzz drivers, research questions, problem statement, research methodology and contributions of the thesis.

- Chapter 2 answers the first research question by reporting the results of our SLR that identifies the background, methodologies and challenges of fuzz driver development.

- Chapter 3 answers the second research question by carrying out a case study to compare the performance of three different types of fuzz driver generation techniques with open-source software products. Chapter 3 also answers the third research question through an empirical analysis to explore how different types of code complexities affect the fuzzing campaign and especially how different types of fuzz drivers behave under different code complexities.

- Chapter 4 concludes the thesis with a summary and potential future directions

We are in the process of finalising a journal article submission based on Chapter 2. Chapter 3 will form the basis for an upcoming workshop paper submission.

1.6 Thesis contributions

This thesis makes the following contributions,

- A Systematic Literature Review of the role of fuzz driver development in fuzzing campaigns (Chapter 2).
  - Identification and analysis of requirements behind fuzz driver development and proposing the steps to develop a fuzz driver effectively.
  - Analysis of different fuzz driver development practices and thorough exploration of techniques adopted by the researchers to formulate the steps to develop a fuzz driver for a maximised performance.
  - Identification of fuzz driver evaluation techniques to identify the best practices that testers should follow for an effective fuzzing campaign.
- A case study to identify the best fuzz driver development strategy (Chapter 3).
  - Design of a case study with a systematic experiment to explore the performance of different fuzz driver generation strategies.
  - Empirical analysis on fuzzer performance metrics during different fuzz driver usage to identify the best state of the art fuzz driver development technique available for software testers and researchers.
- An empirical study to identify the effect of the complexity of the target code on fuzz drivers (Chapter 3).
  - Design of an experimental study to explore how the performance of fuzz drivers generated through multiple techniques fare when they are used against target code of varying complexity.
  - Identification, evaluation and recommendation of best approaches to develop fuzz drivers and to run fuzzing campaigns when the code complexity changes in the target code.

1.7 Other contributions

In addition to the research studies constituting the core of the thesis, I contributed as a co-author for the research publication [130] during the first six months of my research candidature. This research work is not directly related to my thesis, however, it is still in the domain of software testing and software quality assurance. We explored the flakiness (non-deterministic behaviour) of unit tests. We proposed a technique to automatically classify tests as flaky or not based on their vocabulary. On this occasion, I solved this problem using five different machine learning algorithms. I gathered 1000 flaky tests and 1000 non-flaky tests and used machine learning algorithms to

predict the flakiness of the test based on their features. My work achieved an F-score (harmonic mean of precision and recall) of 0.95 for the identification of flaky tests. Furthermore, my work on this research study identified that Random Forest and Support Vector Machine machine learning algorithms give the best results for flaky test prediction.

Chapter 2

Fuzz driver generation for fuzz testing

2.1 Introduction

Due to the introduction of open-source automated fuzzing tools, both small-scale and large-scale companies adopted fuzzing in their software development life-cycle. These fuzzing tools can monitor crashes and collect the metrics related to the crash and the fuzzing process. When it comes to fuzzing software products, a fuzz driver plays an important role because it directs the fuzzer to fuzz certain parts of the code.

Despite the mentioning of fuzz drivers in multiple research studies, there is a lack of deep analysis of its literature; consequently, the role of a fuzz driver is insufficiently understood, as are techniques for its generation, challenges and future directions. This chapter contributes a systematic literature review (SLR) on fuzz drivers.

We used Google Scholar to extract literature for this research study and we filtered data systematically in multiple steps to arrive at 102 papers related to the fuzz driver domain. We critically analysed all of these papers and categorised them into an attribute matrix that consists of five categories. An attribute matrix is a representation of how different themes emerge from research and how different aspects identified through research match these themes. The summaries of these attributes and their subcategories formed the basis of this SLR that explored the importance, techniques, new methodologies and challenges when developing fuzz drivers in fuzzing.

As explained in Chapter 1, a "Fuzz Driver" is the binder between the fuzzer and the system under test. It accepts an array of fuzzed input coming from the fuzzer and sends it to the target function for fuzzing. The fuzzing life-cycle consists of six main steps and writing a fuzz driver is the second step of the fuzzing life-cycle as shown in Figure 2.1.

The purpose of the fuzzer is to generate a set of fuzzed inputs to cause a crash in the target program. General-purpose fuzzers generate unstructured input to carry out this task, however, software libraries expect highly structured input. Software libraries are the files that contain reusable code which can be invoked by another library or an executable. If the unstructured input is sent to software libraries without any mediation, it would generate crashes that would most likely be deemed as false positives. In the context of fuzz testing, a false positive is an incorrect flagging of a software defect that does not exist. Therefore, fuzz drivers fill this gap by acting as an interface between the fuzzer and the target program in the software library to direct the unstructured input correctly to relevant parts of the software library for fuzzing.

From the papers that we included in the SLR, Figure 2.2 shows the distribution of research studies that mention fuzz drivers per year. Even though fuzzing has existed since 1990 through Miller's research [106], the first paper that mentions the concept of fuzz driver appears in 2007 [36]. The number of papers remains relatively constant

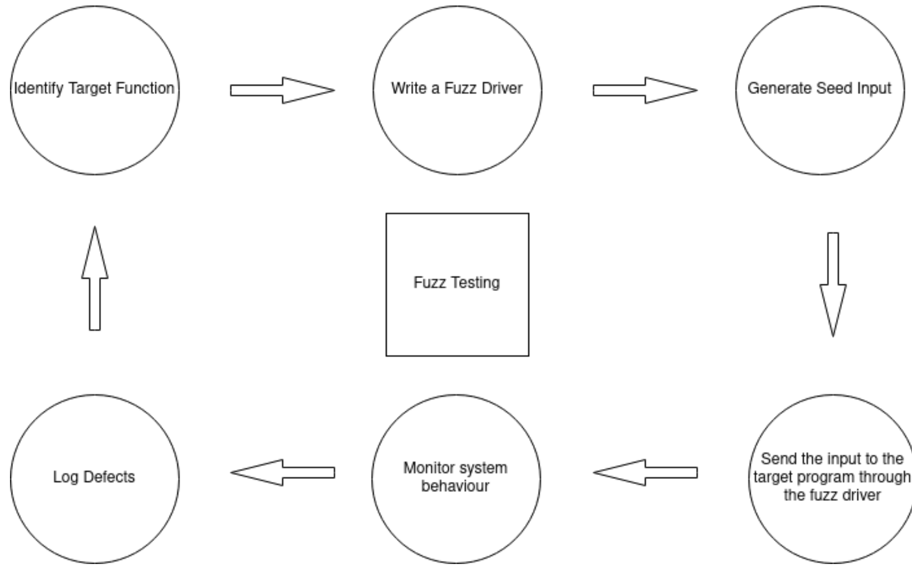

FIGURE 2.1. Fuzz testing life-cycle.

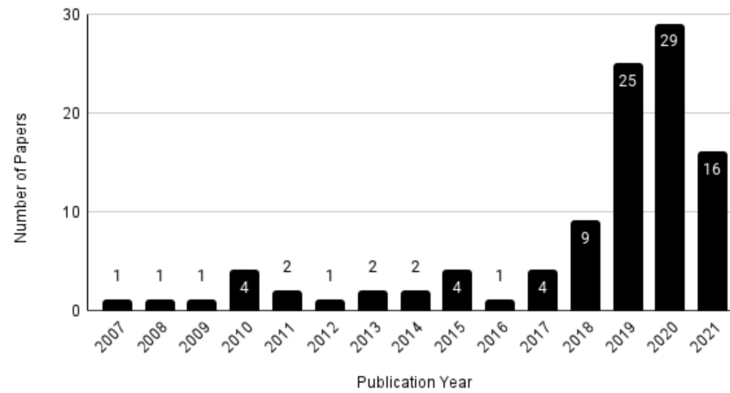

FIGURE 2.2. Number of papers that mentions fuzz drivers vs the publication year.

until 2017, after which it quickly increases to 29 papers in 2020. Our SLR found that those research studies collected from 2018 show the use of fuzz drivers in multiple fuzzing platforms and there is increased attention given to enhancing the practices of fuzz driver development through automation. Hence, this SLR aims to explore this expanding domain and to show its trends and research gaps.

Figure 2.3 shows all 102 research materials that we collected and how different types of fuzz driver generation methods are distributed along with research publications. Papers that mention manually written fuzz drivers are shown in black, semi-automatic methods shown in yellow and fully automatic methods shown in green. Papers that show fuzz drivers that are built-in components (single harness/ driver (sometimes a template) comes with the program), Use of existing fuzz drivers or other materials (unit test) are shown in purple. Finally, papers that mention fuzz drivers but do not contain the method of development are shown in red. It is clear that manually writing the fuzz driver is the most common method for fuzz driver development. Since, 2019, there is an interest in developing semi-automated methods for fuzz driver generation, however, that interest nullifies when it comes to 2021. In 2021, the interest in fully automated fuzz driver development greatly rises compared to

previous years, thus showing the current main attraction in fuzz driver research. The complete list of our collected research materials is available in Table A in Appendix A.

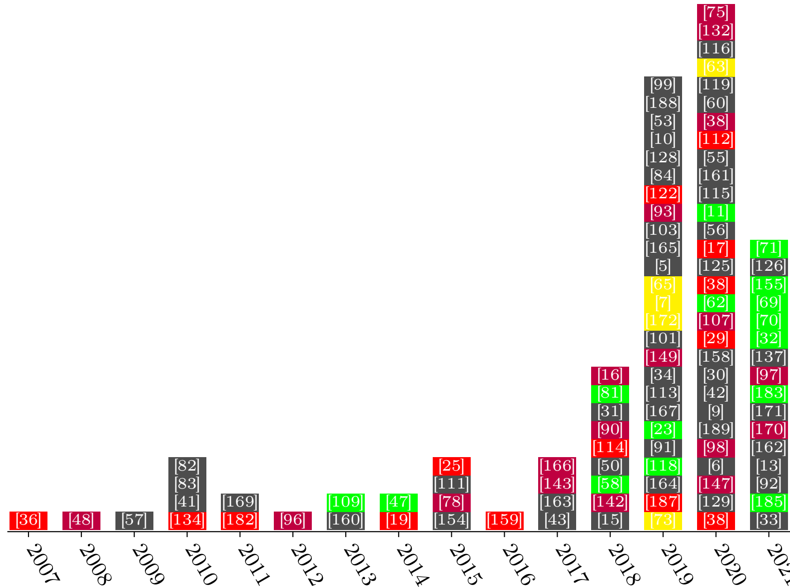

FIGURE 2.3. Distribution of papers along publication years.

There are multiple SLRs written on fuzzing [89, 99, 102] but this is the first SLR written about fuzz drivers. A systematic literature review of fuzz drivers allows researchers and security experts to gain a deep understanding of the common practices and the challenges and limitations of fuzz drivers. It provides a knowledge base in terms of existing techniques in developing fuzz drivers, exploring the potential to automate or semi-automate the fuzz driver development and identification of evaluation criteria for meaningful outcome generation. Furthermore, it gives insights regarding patterns in research, development criteria and novel innovations associated with fuzz driver generation by analysing 102 research papers relevant to this topic and provides new research directions for the research community. Our key contributions are,

- Understanding the reason behind fuzz driver generation (Section 2.6).
- Classification of fuzz driver development practices (Section 2.7).
- Categorisation of fuzz driver evaluation methods (Section 2.8).
- Discussion of identified gaps in research that require enhancement through future research (Section 2.10).

Among other, manually writing a fuzz driver is the most common occurrence in nearly half of the research studies (50 research studies). However, since 2013, there are novel automated and semi-automated methodologies proposed to generate fuzz drivers. When developing fuzz drivers, the authors of scientific articles consider the methods of reaching a bug, types of bugs, the effect of a seed corpus on a fuzz driver, the effect of multi-threading, speed, timeouts, difficulties of manual fuzz driver development and the requirement for automation of fuzz driver development.

Fuzz driver development life-cycle consists of target identification, program analysis and driver synthesis. When a fuzz driver is written manually, both function identification and dependency analysis are done by the developer. However, when the

fuzz driver development is automatic or semi-automatic, researchers use methods such as annotations, abstract syntax trees, header files, program slicing to identify functions. They further use control flow graphs, data flow graphs, data dependency graphs and project control graphs for function dependency identification. These provide the order sequence of API calls for fuzz driver synthesis.

C, C++ and Java are the most commonly used languages. The most commonly used metrics for fuzzer evaluation are the number of bugs, code coverage, fuzzing time, fuzzing speed, identification of a certain bug and number of inputs per crash. The most commonly used fuzz driver evaluation techniques are comparisons of manually and automatically generated fuzz driver, interface model evaluations, the measure of the number of APIs and the comparison with other automatic fuzz driver generators.

The rest of the chapter is organised in the following manner. Section 2.2 gives a brief background on fuzz drivers and their usage. Section 3.3 describes the research methodology followed for systematic data extraction for this research study. Section 2.4 gives an overview of the most commonly used results and findings of the SLR. Sections 2.5 to 2.9 explores these research findings in detail. In Section 2.10, we reflect upon the research gaps and discuss potential future research directions. Section 2.11 explores the limitations and threats to validity and in Section 2.12, we summarise this research study.

2.2 Background on fuzz drivers

The role of a fuzz driver is to send random input generated by the fuzzer to the specific functionality of the code that requires fuzzing. Some of the literature refers to a fuzz driver using the terms "fuzz target" or "fuzz harness".[1] It is not uncommon for fuzz drivers to be linked with multiple fuzzers such as LibFuzzer [94], American Fuzzy Lop (AFL) [179], HonggFuzz [59].

Researchers and software testers use different fuzzers to fuzz code. LibFuzzer is one of the fuzzers that allow fuzzing software libraries [139]. When developing a fuzz driver for LibFuzzer, software testers should define a function called `LLVMFuzzerTestOneInput` with a predefined signature, see the example in Listing 2.1. On this occasion, the target function is an array function. This function accepts two parameters to fuzz the target library. The first parameter is an address of a buffer that accepts random data generated by the fuzzer. The second parameter is the size of the data in the buffer.

It is the task of the software tester to write the body of this function according to the functionality of the target program. When writing the fuzz driver, parameters of that particular function should be combined with library API functions. To carry out this task, the developer has to initially select a target function to pass input from the fuzzer to the library, which is a function that contains instructions to load a value to the memory to be used by a library. Software testers should use library functions to test multiple features of the library.

[1] From the data that we collected, "fuzz harness" is the most common term with 52% followed by "fuzz driver" 38% and "fuzz target" 10%. However, the terms "fuzz harness" and "harness" are similarly used to show test benches and testbeds. To avoid ambiguity, in this work, we consistently use the term "fuzz driver".

Listing 2.1. Example LibFuzzer fuzz driver.

```
int LLVMFuzzerTestOneInput(uint8_t * fuzz_input_data, size_t
    fuzz_data_size) {
     size_t fuzzer_input_min_size = 0;
     if(fuzz_data_size < fuzzer_input_min_size) return 0;
     uint8_t * fuzz_ptr = fuzz_input_data;
     int * arr = (int *)fuzz_ptr;
    int n = (fuzz_data_size - fuzzer_input_min_size) / sizeof(int);
     (void)fun(arr, n);
     return 0;

}
```

Fuzz driver generation is a time-consuming process and requires intricate knowledge of the codebase [7, 23, 50, 62, 65, 73, 90, 91, 114, 135, 142, 147, 149, 182, 186]. As a result, starting from mid 2010s, multiple research studies started identifying ways to automate this process. In the following sections, we report fuzz driver development methods and techniques with insights in more detail.

2.3 Methodology

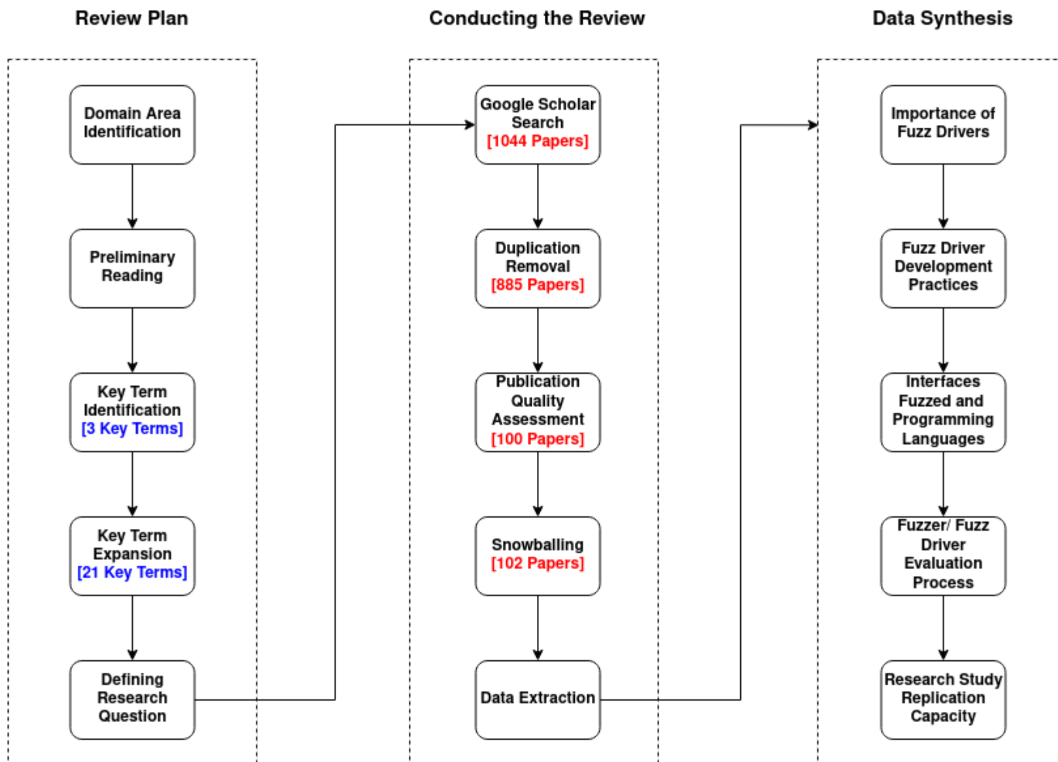

Figure 2.4. Overview of the methodology.

This section describes our surveying methodology for identifying publications and extracting information. In a nutshell, we followed the systematic approach of Webster and Watson [173] and Kitchenham [79]. In the rest of the section, we describe the steps of our systematic approach: research questions, search strategy, paper filtration criteria, snowballing and synthesis of results.

- Research question: **What are the fuzz driver generation practices adopted by researchers?**

Through this research question, we intended to identify how fuzz drivers are mentioned in research papers. We identified fuzz drivers in terms of their purpose and how beneficial it is to the process of fuzzing. Similarly, other themes such as program languages used in fuzz driver development, how fuzzers support fuzz drivers and the coding practices involved when writing a fuzz driver emerged from our analysis of papers. Furthermore, we categorised fuzz driver development strategies along with their evaluation criteria.

Search strategy: Miller et al. [106] published the first-ever paper on fuzzing in 1990, thus this was the starting point of our study collection. We used Google Scholar to extract research material for this SLR since it provides full text and metadata of scholarly literature across an array of publishing formats and disciplines. There are multiple research databases available to extract research papers, however, large scale searches on the metadata of research articles in multiple research databases are different due to different search algorithms adopted by their online systems. We ran an array of search terms to extract results, therefore we observed the consistency in terms of returned results for the same search term across all the platforms. Moreover, for this research study, we included grey literature. Grey literature includes unpublished studies (e.g. pre-prints) or literature published in non-commercial form (e.g. PhD thesis). Grey literature is very important for an SLR to remove publication bias during the paper extraction stage for any given topic [177].

According to Yasin et al. [177] Google Scholar is a great source to extract primary studies to include grey literature in the SLR compared to other research databases. Furthermore, Google Scholar indexes research from multiple research databases. Therefore, we gathered all the scholarly results related to fuzz drivers including grey literature and this includes sources such as Pre-Prints, Masters and PhD theses that would not appear in research databases such as "IEEE Xplore" if they are not published at one of their sponsored events. However, these results were available to us through Google Scholar. Hence, to include grey literature following the work produced by Yasin et al. [177] and to keep the consistency, we used Google Scholar to extract data sources for this SLR.

After reading a set of papers that were written post 2010 regarding fuzz drivers and fuzz driver development [7, 11, 31, 60, 62, 65, 69, 71, 73, 91, 119, 129, 183], we identified a set of terms that we would use to filter research papers in the domain of fuzz drivers. As a result, we identified multiple search terms for the article retrieval process. We further consulted the documentation [94], books [180], and literature surveys [89, 99, 102] related to fuzzing to clarify these key terms.

We first identified the terms "fuzz driver" [7], "fuzz harness" [116] and "fuzz target" [73] as the most common terms that defined fuzz drivers. We then systematically added possible extensions (Table 2.1) to each of these to identify further results. Specifically, we identified the use of the term "fuzzing" as an alternative to "fuzz" [71]; that the words "test" or "testing" are sometimes added to the term [74, 148, 160]; and that these are sometimes used as alternatives to "fuzz" or "fuzzing". Therefore, after processing these key terms, we identified 21 key terms to search Google Scholar to extract relevant papers that would contain research related to fuzz drivers. Table 2.1 summarises the search term with identified keywords.

Google Scholar search strings produced a large number of duplicates due to overlapping results. Hence, as the next step, we removed duplicates, reducing the

TABLE 2.1. Search terms and number of Google Scholar results.

Term	Results	Term	Results	Term	Results
"fuzz target"	63	"fuzz driver"	20	"fuzz harness"	8
"fuzz test target"	0	"fuzz test driver"	2	"fuzz test harness"	1
"fuzz testing target"	1	"fuzz testing driver"	0	"fuzz testing harness"	3
"fuzzing target"	120	"fuzzing harness"	19	"fuzzing driver"	17
"fuzzing test target"	1	"fuzzing test driver"	1	"fuzzing test harness"	0
"test target" fuzz	199	"test driver" fuzz	187	"test harness" fuzz	267
"testing target" fuzz	77	"testing driver" fuzz	15	"testing harness" fuzz	43

paper count to 885.

Paper filtration criteria: We used a set of inclusion and exclusion criteria to filter out papers for this research study.
The inclusion criteria are,

- The research paper should be written about Fuzzing.
- At least one of the terms explained in Table 2.1 should be mentioned in the research paper.
- The document should be written in English.

The exclusion criteria are,

- Different interpretation of the terms "fuzz driver", "fuzz harness", "fuzz target".
- The research paper is not in the domain of Computer Science.

The assessment of the inclusion of the papers was mainly dependent upon their capability to answer the research question and their usefulness in developing conclusions for the literature survey [72]. When it comes to research papers, we initially checked their relevance to the domain of fuzzing by reading the paper abstracts of 885 papers. Then we checked whether it mentioned the use of fuzz drivers by reading the aims and objectives, methodology and the conclusion parts of the paper. If it is directly related to the fuzz drivers or if they mention the use of fuzz drivers in the research study, we collected that paper from the pool of our sources. Additionally, we read the parts of the papers where the above keywords appeared to confirm their relevancy to the domains of fuzzing and fuzz drivers: if the paper mentions fuzz drivers, it was kept. Otherwise, it was discarded. At the end of this procedure, we reduced the paper count to 100 papers from the sample of 885 papers.

Snowballing: To expand the extracted research studies, we carried out both forward and backward snowballing [176]. This process increased our paper count by two papers [65], [99], increasing the final study count to 102. These 102 studies contain conference papers, journal papers, book chapters, documentation, workshop tutorials and Undergraduate, Masters, and PhD theses.

Synthesis of results: After the paper selection, we analysed and categorised them in terms of information related to fuzz drivers and fuzz driver generation by following the narrative of the research question. We went through multiple iterations of discussions to identify the most commonly used themes and refine these and develop the attribute matrix. Since our research question covers a broad area, we initially started the research study analysis by looking at the importance and the reasoning behind fuzz driver development.

We observed "Why" such a concept is required for fuzzing. The next step was to identify fuzz driver development practices. We monitored "How" to develop fuzz

drivers for various circumstances and categorised those methodologies. In the next few iterations, we further monitored systems that require fuzz drivers for fuzzing and the programming languages required to develop these drivers. Furthermore, we extended the categories to identify which percentage of papers made their code publicly available for results replication and which percentage of papers included code snippets of fuzz drivers within the text of the paper. Finally, we further extracted fuzzer and fuzz driver evaluation metrics. When we read these collected papers, we observed common themes. As we collected information regarding these themes, more directed themes regarding fuzz drivers started to emerge from texts. These themes are,

- Context of fuzz driver mentions in fuzzing research (Section 2.5)
- Requirements for fuzz driver development (Section 2.6)
- Fuzz driver development practices (Section 2.7)
  - Different methods of fuzz driver development
  - Fuzz driver generation steps
  - Source code analysis for target selection
  - Function dependency identification
  - Interfaces fuzzed
- Fuzzer and fuzz driver evaluation methods (Section 2.8)

Figure 2.4 shows the summary of the data gathering process that we followed out systematically with the number of resulting research papers.

2.4 Findings

We present our findings in terms of the themes that emerge from the research studies. They are the context of fuzz driver mentioned in fuzzing research, the requirement behind fuzz driver development, fuzz driver development practices and fuzzer and fuzz driver evaluation methods.

2.4.1 Context of fuzz driver mentions in fuzzing research

We identified the importance of fuzz drivers in terms of observing the depth in which it is mentioned in research studies. Fuzz drivers are expressed in multiple ways. The majority of the papers mention the development of a fuzz driver as a support activity in fuzzing projects by giving it minor attention in a larger scope. A subset of these papers show novel approaches while appearing as minor methods in papers that fuzz drivers are not the main focus of research Conversely, other papers propose novel methods that solely focus on fuzz driver development as the main direction of their research studies. Another set of papers mentions fuzz drivers with no novelty but as extensions of previous novel work. Furthermore, two papers introduce a method to bypass the fuzz driver development process. These findings are explained in Table 2.4 and further explored in Section 2.5.

It was evident from Figure 2.1 that until 2018, there is less focus on fuzz drivers in research compared to the period from 2018 to 2021. Until 2017, novel methods are introduced only three times [47, 109, 134]. However, there is a rapid increase in fuzzing research from 2018 with the development of novel methods to automate fuzz

driver generation, hence it could be regarded as one of the reasons for the increase of papers mentioning fuzz drivers post-2018. According to our findings, two of the highly used fuzzers in fuzz driver-related research are AFL and LibFuzzer. Before 2016, the majority of research studies show their fuzzers were developed to match their research criteria rather than using a single open-source product for fuzzing. Both of these fuzzers require fuzz drivers to function. AFL was introduced in 2014 [179] and LibFuzzer was introduced in 2015 [94]. Our results show that the first paper that mentioned AFL appeared in 2016 [159] and the first paper that mentioned LibFuzzer appear in 2018 [15]. From 2018 onward, these two fuzzers make a high appearance in research papers, hence it implies that the introduction of open-source software such as AFL and LibFuzzer, increased research interest in fuzz driver research.

2.4.2 Fuzz driver requirements

Factors such as deterministic behaviour, fast operation and avoiding intentional crashes are the main required behaviour of an efficient fuzz driver [9]. Identifying the nature of the required input values to a function using its function signature plays a major role in developing meaningful fuzz drivers [78]. The requirement to automate the fuzz driver development is a recurring aspect in research studies as explained in the previous section Section 2.4.1. SLRs on fuzzing showcase an exponential increase in fuzzer related research over the years from 1990 [89, 99, 102]. However, the first fuzz driver-related research appeared in 2007 [36] and our results do not show similar exponential growth in interest in fuzz drivers. It seems to be the highly focused research around AFL [179] and LibFuzzer [94] initiated this rise since both of these products rely on fuzz drivers and researchers tried to look for ways to make their development process more user friendly.

It is important to have high code coverage to increase the chance of identifying a bug when attempting to identify a bug through fuzzing [154]. We recognised the importance of identifying as many paths as possible to cover hard-to-reach parts of the code. It is also important to identify program states and control/data flow information. The literature covers types of bugs such as buffer overflows, stack overflows, and segmentation faults. Another research interest is the consideration of garbage collection (Section 2.6.6) for memory management and its effect on fuzz driver to maintain memory management [73]. More information about these aspects with example results is given in Section 2.6. Using this information, we evaluated how these identified factors affect the fuzz driver development process in Section 2.7.

2.4.3 Fuzz driver development practices

There are four methods of fuzz driver development in research. They are,

- Fuzz driver development as a manual activity that requires input from a software developer.
- Fuzz driver development as a semi-automatic approach (automatic with human-in-the-loop techniques).
- Fuzz driver development as an automatic approach.
- Fuzz driver development using approaches such as use of built-in-components, pre-existing fuzz driver or a conversion of a unit test into a fuzz driver to aid the fuzzing process.

Our results show that manually developing the fuzz driver is the most common practice while fuzz driver automation seems to take high interest from researchers during the last four years with research papers that are solely dedicated to fuzz driver development being written to improve fuzz driver development methodologies [7, 62, 71, 73]. The breakdown of these methodologies is explained in Table 2.4 and we explored these techniques in detail in Section 2.7.

The first step of the fuzz driver development process is to identify functions and function signatures in the target program that require fuzzing. The software developer has to identify the required functions and their dependencies to other functions to fuzz using their domain knowledge when they write a fuzz driver. Human involvement is required to identify target functions during the manual and semi-automatic fuzz driver generation. It is important to identify function dependencies When multiple functions require fuzzing through a single fuzz driver. Furthermore, there are occasions where the reuse of fuzz drivers occurs. These multiple challenges are explained in Section 2.6.10. It is evident from our results that writing a fuzz driver by a software developer is a challenging task due to reasons such as time consumption, lacking a deep understanding of the program domain and human error.

When research studies adopted automated methods, some carried out program analysis to identify functions to fuzz using multiple methodologies. This involves techniques such as the use of abstract syntax tree (AST) extraction [7, 109, 185], program slicing [7, 58, 78], inclusion of annotations [73], unit test analysis [90, 91, 93], LLVM IR generation [62] and type signature scanning [62, 186]. These function identification techniques for fuzz driver generation will be explained further in Section 2.7.2.

After the function identification, function dependencies play a major role in identifying program flow for the function synthesis. The software developer plays the role of identifying function dependencies for manually developed fuzz drivers. A large amount of the papers (50 out of 102 papers) show function dependency identification as a manual process. On some occasions, function dependency is not monitored for fuzz driver generation. This applies on both occasions where one function is fuzzed [60] or selected functions are fuzzed [73]. On some occasions [11] all the functions are considered for fuzzing thinking that every function is a potential entry by ignoring function dependency analysis.

Furthermore, when fuzz driver generation through unit tests [65] or the reuse of existing drivers, researchers ignore dependencies since they already exist in reused unit tests and fuzz drivers. Major trends in identifying function dependencies in automated fuzz driver development are API dependency graphs, control flow graphs, data flow graphs and project control graphs. Identification of function priority order aids the function dependency identification. Techniques used to extract information regarding function priorities are API call sequence extraction [68], trace capture [70] and user-defined function (UDF) coverage [186]. We explain these ideologies in detail in Section 2.7.3.

There are multiple programming languages and coding practices used in developing fuzz drivers. It is a recommended practice to make the source code used for experiments publicly available since it promotes experiment replication [86, 127, 151]. From 102 research studies, only eight studies contain open source projects. All of these eight open-source projects are from research studies carried out after 2018 and five of those eight projects are based on AFL and LibFuzzer. This further solidifies our argument from Section 2.4.2 that the introduction of AFL and LibFuzzer had an effect to increase interest in fuzzing research and in particular increasing interest in fuzz driver development. We explore the availability of open-source code, programming languages and the availability of fuzz drivers in the text of the research studies

more thoroughly in Section 2.7.4.

2.4.4 Fuzzer and fuzz driver evaluation techniques

We explored the evaluation metrics of fuzzers and fuzz drivers. We explored fuzzer evaluation on this occasion since the fuzz driver is a component of the fuzzing process. The most commonly used fuzzing evaluation metrics are measuring the number of bugs, code coverage, fuzzing time, fuzzing speed, identification of a certain bug and number of inputs per crash. When it comes to fuzz drivers, researchers compare manually and automatically generated fuzz drivers, interface models, a measure of the number of identified APIs and comparisons with other automatic fuzz driver generators. The number of bugs and the code coverage are the two main metrics used to identify the effectiveness of fuzz drivers of these multiple fuzz driver generation techniques. We explain these methodologies in detail in Section 2.8.

2.5 Context of fuzz driver mentions in fuzzing research

Researchers mention fuzz drivers in multiple contexts in their research papers. When it comes to the development of fuzz drivers from various projects since 1990 [106], they discuss fuzz drivers in multiple ways. We categorised those into five categories (Table 2.2),

Table 2.2. Prioritisation of fuzz driver generation in research papers.

Prioritisation	Count
Minor task in a larger scope (developed by software testers)	79
Novel method but not the main focus of the study (fuzz driver developed as a sub process)	8
Introduces a novel method directly focusing on fuzz driver development	7
Extension from previous work for fuzz driver generation	6
Novel method to avoid fuzz driver generation (bypassing fuzz drivers)	2

There are 79 papers that discuss fuzz drivers as a small part of a large fuzzing project. This means that their focus is on a different aspect of fuzzing and they discuss the use of fuzz drivers developed by software testers as a minor part of their research. These papers mention the manual input required by software testers and the need for a fuzz driver written by a developer for fuzzing. For example, Cai et al. [19] explores writing fuzz drivers by software testers for C binaries. Ding and Goues [33] focuses on empirical analysis of OSS-Fuzz bugs and they briefly mention the use of fuzz drivers as entry points for the fuzzing process. Frighetto et al. [42] talks about software testers writing fuzz drivers for C and C++ programs and fuzzing those programs through LibFuzzer to identify important vulnerabilities. Furthermore, Pham et al. [129] discusses the use of fuzz drivers to Fuzz C programs using AFL as the Fuzzer of choice.

Similarly, eight papers [11, 58, 65, 70, 93, 109, 118, 185] show the process of generating fuzz drivers through a novel method but fuzz driver generation is not the main focus of these papers. Holland [58] proposes the use of the Mockingbird framework [108], which can enable targeted dynamic analysis of the code base through the use of binary inputs. It focuses on Java programs and by using the Mockingbird framework, binary inputs can be transformed into relevant object types. Mockingbird allows the automated generation of fuzz drivers.

Similarly, Zhang et al. [185] proposes a tool called BigFuzz which uses fuzz drivers in Java program fuzzing. BigFuzz subdivides the program into six sections based on

User Defined Functions (UDF) which contain the name of the operator and the order of the execution. BigFuzz automatically generates a fuzz driver to aid the fuzzing process using the execution specifications of these Java classes. Furthermore, FuzzBuilder [65] automatically generates fuzz drivers by carrying out static and dynamic analysis of already existing unit tests written for the codebase by automating the fuzz driver generation process. Note that all of these papers discuss fuzz drivers as a minor process to support the main topic and findings of the research study.

Seven papers [7, 62, 69, 71, 73, 81, 183] introduce novel methods that aid the process of developing a fuzz driver. These studies focus on enhancing the state-of-the-art development methods by automating manual practices carried out by software testers. They have a direct focus on finding a novel way to enhance the fuzz driver generation process. Research studies such as FuzzGen [62], Fudge [7], Intelligen [183], Fuzz Target Generator (FTG) [73] describe automating the fuzz driver development by minimising human input. Some of these research studies automate the development of fuzz drivers entirely [7, 62, 183] while others follow a hybrid approach [73]. We will explain these techniques in detail in Section 2.7.

Moreover, another six papers [25, 30, 63, 78, 107, 155] discuss the fuzz driver generation process as an extension of the work that they have done previously. Miyaki et al. [107] mentions the reuse of AFL fuzz drivers to evaluate their project and, similarly, Cummins [30] shows the reuse of fuzz driver generation method for kernel fuzzing.

Finally, two papers [11, 47] proposed novel methods to avoid fuzz driver generation. These papers are trying to find ways to bypass the creation of fuzz driver generation. Godefroid [47] promotes micro executions to avoid the use of fuzz drivers and then later Blair et al. [11] extends on this work. This will be further analysed in Section 2.7.3. Fuzz Driver generation is characterised through this categorisation, which allows us to understand its context in this research. Those papers that solely propose a novel approach to minimise or nullify the manual development of fuzz drivers discuss the development of fuzz drivers as its main focus. On the other hand, those who develop fuzz drivers manually give it minimal attention.

2.6 Requirements for fuzz driver development

We explored the motivation behind fuzz driver development and the challenges that they implicate in its development process and how they would affect fuzzing.

2.6.1 Characteristics of a fuzz driver

In this section, we describe our findings related to the characteristics that a fuzz driver needs to have to work effectively with the target program.

Deterministic behaviour: When fuzzers such as LibFuzzer and AFL are carrying out the fuzzing campaign, the goal is to execute the target program as quickly as possible with a help of a fuzz driver and a good seed corpus. There are certain characteristics that fuzz drivers should avoid to increase the speed of the fuzzing campaign and to create reproducible crashes. First of all, a fuzz driver should be deterministic [9]. To promote this behaviour, the fuzz driver should not rely on specific multi-threading behaviour or random number generators; it should have the ability to replicate the same crash multiple times consistently for the same input. When fuzz driver in DeepState is created for testing binaries, they build the constructs using variadic templates [50]. They claim that this allows the execution of code chunks in

a deterministic manner to reach the fuzzing standards. Furthermore, the work done by Fioraldi et al. [39] proposes modifications of fuzz drivers to set an environment variable to preload the run time to deal with the deterministic behaviour of the fuzzer.

Fast operations: Fuzz drivers should also avoid slow operations [9]. Software testers should focus on developing memory-based file systems and fuzzer-friendly builds that avoid and disable these slow operations. Chen [23] explains that writing a fuzz driver is a manual task. when there is a lack of source code, there is a high overhead caused by dynamic emulation, which affects the speed.

Avoiding intentional crashes: Moreover, there is a requirement to write fuzz drivers in a way that they avoid intentional crashes [9]. This is because the main goal of fuzzing is to identify unknown vulnerabilities and vulnerabilities that the developer did not purposely intend to have at the programming phase rather than observing known vulnerabilities.

2.6.2 Function identification

To identify which functions to fuzz, it is imperative to identify types of function signatures since the functionality of the fuzz driver should match the function signature types of the target function. This is because the function signature shows the acceptable input to the target function. Most papers follow a methodology where the developer manually finds the function signature and writes the fuzz driver as mentioned in [15, 103, 113]. However, some papers introduce human in the loop approaches such as annotations [73]. Also, there are other methods such as the generation of configuration files by analysing code [58] and scanning through C header files [62] to identify function signatures. We will explain these techniques further in Section 2.7.2.

2.6.3 Path identification

It is important to identify the paths of the program when writing a fuzz driver. Knowing multiple paths will make it easy to reach multiple parts of the program. Maier et al. [98] explains the triggering of multiple paths when the fuzzing in the target program. Wang et al. [171] says that the difficulty of path identification increases when the lines of code increase. They claim that manually developed fuzz drivers in Google's fuzzer-test-suite are designed by recognising and identifying this aspect along with other important factors, hence it can trigger the known CVE when the fuzzer provides correct input.

According to Oleksenko et al. [119], when an assumption is made for the maximum depth of the speculative execution to become 250 instructions, the number of speculative paths increase up to 30 million for an average conditional branch. However, they suggest that the number of actual branches is lower in value due to imbalance in the tree or the shallowness in the tree due to serialisation instructions (e.g. System Calls). Oleksenko et al. [119] believe that serialisation instructions cause the fuzz driver to slow down by order of magnitude. They claim if it is a small fuzz driver for a single function in the library, this problem is not an issue, however, if the fuzz driver is targeting a larger library, then it requires rectifications by the software tester.

Reaching hard to reach components of the code

Fuzz drivers allow the input seed to reach multiple sections of the code for fuzzing; fuzz drivers guide the input seed to hard to reach sections of the code [154]. The research

from Shastry [142] claims the difficulty to achieve 100% API coverage through fuzz drivers and the requirement to adopt static analysis to enhance the code coverage metrics.

Furthermore, Ognawala et al. [118] also mentions that fuzz drivers should target isolated functions. When fuzzing using the DeepFuzzer [91], it fails to identify a known memory leak vulnerability due to the lacking of system knowledge when developing the fuzz driver. Similarly, AssetFuzzer [83] fails to identify certain bugs due to its failure to detect all atomic-set serializability violations in programs that are multi-threaded. Similarly, SpecFuzz [119] fails to reach all the code paths with the existence of the fuzz driver. Hence, these study results entail the requirement for further improvements to increase code coverage to aid the bug identification process.

2.6.4 Program state and control flow identification

The program state and the control flow are important when it comes to fuzz driver generation. The program state is communicated to the appropriate function through the stack memory by using function parameters and return values or through the use of heap memory that contains information regarding the reads and writes to the global variables [58]. Static analysis promotes the identification of the flow of the program [5]. Jung et al. [71] entail that the identification of program flow through the call sequences, control flow and data flow graphs are vital processes in the fuzz driver development process in their proposed tool, Winnie. Furthermore, it was evident about the use of control/data flow graphs for program flow identification when generating fuzz drivers. We analyse these further in Section 2.7.3.

Enhancing the code coverage in fuzzing is directly related to path identification, reaching hard to reach aspects of the code, program state and control/data flow graph identification. The quality of the fuzz driver plays a potent role in this process and this has repeatedly been mentioned in research papers along with fuzz driver creation. Our results show that 27 papers discussed the requirement to increase code coverage along with fuzz driver creation.

2.6.5 Types of bugs

Fuzzing identifies many types of bugs. In particular, commonly mentioned bugs are buffer overflows, heap overflows, stack overflows, segmentation faults, deadlocks and other memory corruption bugs. Many research studies such as [5, 36, 91, 163] specifically talk about the prevention of memory leakage issues in the code using fuzzing and the use of a fuzz driver to achieve these criteria.

2.6.6 Garbage collection in fuzzing

Garbage collection is a way of automatic memory management where it focuses on the process of reclamation of allocated computer memory. Memory that is no longer referenced is defined as garbage in this instance. Memory management is closely monitored with fuzz driver generation since fuzz drivers sometimes tend to deal with malloc functions and other memory-related matters. When researchers discuss fuzz driver generation, the process of garbage collection also appears in research along with memory management. Kelly et al. [73] proposes the garbage collection feature in their semi-automatic fuzz driver generation method through a specially defined annotation that would generate code in the fuzz driver to do garbage collection and carry out automatic memory management. Maier et al. [98] mentions the use of garbage collection through their tool to aid the fuzzing process and Liang et al. [91]

shows that garbage collection is an important aspect of memory management for fuzz driver generation.

2.6.7 Importance of the seed corpus

Furthermore, the process of creating an input seed for fuzzing should closely monitor the nature of the fuzz driver. The method of feeding the seed to the system under test solely relies on the identification of the system interface, hence the importance of a good fuzz driver to drive the fuzzer loop. Moreover, the seed corpus should trigger relevant crashes to identify malicious bugs rather than produce false positives. Furthermore, the seed corpus must accommodate the conditions defined in the fuzz driver to prevent false positives [15, 73, 91, 129].

2.6.8 Accommodating parallel fuzzing and multi-threading

Fuzzing done by LibFuzzer is single-threaded, however, it recommends running multiple fuzzing processes parallel with a shared corpus directory since this would allow sharing of any input from one fuzzing process to be available to multiple other fuzzing processes [94]. Therefore, the use of multiple CPU cores quickens bug identification. The fuzzer itself can control parallel fuzzing or multi-threading. Many research discussing fuzz drivers discuss multi-threading and parallel fuzzing to quicken the bug-finding through fuzzing [30, 40, 82, 83]. However, it is important to note that fuzz drivers have no effect on enhancing the multi-threading and parallel fuzzing and it is solely dependent on the type of fuzzer.

2.6.9 Speed and timeouts

Research studies discuss the fuzzing speed mainly in terms of executions per seconds [9]. The quality of the fuzz driver would affect the executions per second for a Fuzzer and the efficiency of the fuzzing process. Also, the speed is related to the coverage of the fuzzer and we explore this further in Section 2.8.

The work from Van Looy [165] shows the fuzzing of software libraries using AFL with a help of a fuzz driver. On this occasion, they have set a timeout limit to one hour; they claim that this is the effective upper bound of time to identify a bug in this library using AFL. Also, it is important to identify the difference between a timeout or an actual crash on some fuzzers. For example, another study proposes a JavaScript driver to aid LangFuzz; they assume one of the reasons for the termination of the fuzzing process is caused by a timeout when waiting for a crash caused by a bug [48]. Hence, it is an important factor to identify a timeout from a crash to recognise a bug in a system.

2.6.10 Difficulties of manual fuzz driver development and the need for automation

Our findings show that 20 papers directly quote the difficulty of manually writing fuzz drivers. These 20 papers claim that the writing of a fuzz driver is a time-consuming task and the need for high expertise from software testers with good domain knowledge.

There are 26 papers directly quoting the requirement for automating the fuzz driver generation by minimising manual efforts. Furthermore, the simplification of fuzz driver writing is another proposal since the manual analysis of the system under test to write a fuzz driver is a rather tedious process [111, 158]. On most occasions,

fuzz drivers target a single fuzzer or programming language. However, it is possible to write a fuzz driver that can be compatible with multiple fuzzers [9, 30, 92]. Moreover, Louridas [96] discusses a method of targeting code from ten programming languages using a single driver.

Furthermore, the practice of reusing fuzz drivers is existent in research. Google's fuzzer-test-suite contain real-world libraries of twenty-eight practical projects [153]. The fuzzer-test-suite contains carefully developed fuzz drivers for each project to trigger the required bug for the correct malformed input. Hence, multiple projects reuse these open-source fuzz drivers to replicate known bugs (e.g. HeartBleed bug).

There are 20 research papers that portray automated and semi-automated methodologies to generate fuzz drivers; we explore these further in Section 2.7. Hence, it is evident that manually developing a fuzz driver is the most common practice; however, there are attempts to automate this task by making it less of a burden on software testers. In the next sections, we will explore these ideologies in intricate detail.

2.7 Fuzz driver development

Since 1990, starting with Miller et al. [106], a wide variety of research has been carried out in the domain of fuzzing and fuzz drivers. Our results implicate that the first paper that mentions the use of fuzz drivers is written in 2007 [36]. It is evident that the first attempt at the automation of the fuzz driver generation process took place in 2013 [109] and as a result, there are multiple papers with automation approaches for fuzz driver generation from 2013 to 2021. This section explains the methodologies adopted by researchers and practitioners to develop fuzz drivers in manual, automatic and hybrid techniques (Table 2.3).

TABLE 2.3. Fuzz driver development strategies.

Technique	Paper Count
Manual development of fuzz drivers by software testers	50
Automatic generation of fuzz drivers	15
Hybrid Method (manual and automatic)	5
Built in Component- single harness or template	18
Unknown	14

2.7.1 Fuzz driver generation steps

Our results show that there are three most commonly used steps to follow when it comes to creating a fuzz driver. The first step is to identify which functions require fuzzing from all the functions in the target program. After identifying the list of functions, the next step is understanding the logic and flow of the program and selected functions. Then using this information, we can synthesise a fuzz driver (Figure 2.5). The following sections show multiple ways of achieving this set of steps in detail.

2.7.2 Source code analysis for target selection

In this section, we attempt to evaluate the initial process of fuzz driver development: the target function identification. Research studies that we gathered show multiple methods for source code analysis. We categorise this process in terms of function

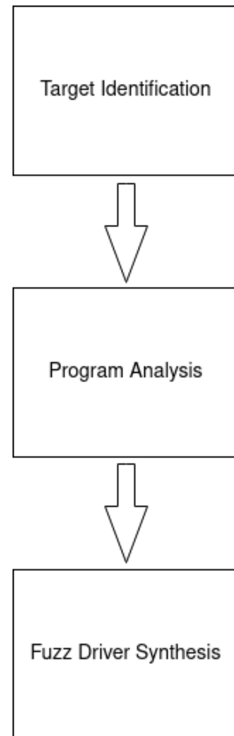

FIGURE 2.5. Stages of fuzz driver development process.

selection by a developer, use of annotations, use of existing unit tests, abstract-syntax tree (AST) extraction, program slicing and function signature monitoring.

Human interference: target identification by a software developer

A large number of research studies show the development of fuzz drivers as a manual process done by software testers (50 research studies) (Table 2.3). Our results showed that 58 studies carries out function identification through a software tester. This shows a further increase from fifty research studies, which shows that certain semi-automated fuzz development methods [73] still relied on careful analysis of the target code by a software tester. These studies require the development of fuzz drivers in various contexts and on multiple programming languages such as C [75], C++ [188], Java [115] and across multiple fuzzers. Deep domain knowledge is required when manually identifying functions [55, 113, 142] and it is a time-consuming process [62, 65, 73, 90]. However, those fuzz drivers with human input show better performance when compared with automated or semi-automated methods except on two occasions [62, 183].

Annotations/code comments

Annotations/code comments aid function identification. This was evident in the research study of Kelly et al. [73] that carries out semi-automatic fuzz driver generation. Only this research study shows the use of annotations for target identification in the fuzzing process, however, annotations do appear for fuzzing related activities in other contexts [115].

The research work by Kelly et al. [73] involves a human tester writing specified annotations on top of the function. This annotation technique appears in automatic unit test generation, Randoop [121]. We believe that Randoop is

the inspiration behind the study of [73]. The tool uses the commented text of `@fuzztest` to generate the fuzz driver. Furthermore, the tool uses additional directives such as `Array (array-ptr, array-len)`, `Value (parameter, value)`, `Output (parameter)`, `Cleanup (condition, function, [params])` to modify the behaviour of the fuzz driver.

Another similar study that uses annotations is FAST [49]. FAST produces test drivers to measure test coverage and not fuzz drivers, hence it is not in the domain of fuzzing. However, this approach has some similarities to Kelly et al. [73] work. It uses ANSI C Specification Language (ACSL) specification, which is a Behavioral Interface Specification Language (BISL) for annotations. It follows the concept of design by contract through the annotations of pre/post conditions, assertions and invariants [49]. These ACSL annotations contain formal first-order logic formulas. Furthermore, these ACSL comments appear in a form of special comments such as `/*@` and `*/` [49].

HyDiff by Noller [114] uses annotations along with fuzz drivers but those two processes are not interconnected. They develop the fuzz driver manually with human input and annotations to drive a symbolic execution process, which is part of the testing process without any effect on the fuzz driver.

Unit tests

Unit tests aid in identifying target functions that need fuzzing. These unit tests are test cases developed by software testers to test the code. Five papers [65, 90, 91, 93, 109] propose the use of unit tests to generate fuzz drivers. These studies propose both manual and semi-automatic approaches. On some occasions, the fuzz drivers would be developed automatically by converting a unit test, however, the unit test itself has to be developed by a software tester. These studies claim that a unit test does have a similar interface compared to a fuzz driver and therefore unit tests can identify the function signature of the target program. Hence the unit test could be modified to be reused for fuzzing in a form of a fuzz driver.

The research work of Liang et al. [90] discusses the requirement of a fuzz driver for an open-source software library called “libmsg”, which allows message exchanges. However, the lack of training to develop fuzz drivers in software testers causes this to be a difficult task even though they have domain-specific knowledge. The fuzz driver for libmsg should have the ability to initialise the library and register callbacks to exercise interfaces. The main component required is a simple request sender to send data from the fuzzer to the library. Hence, Liang et al. [90] believe that the conversion of unit tests and sample code snippets into a fuzz driver takes less effort compared to building a fuzz driver from scratch. Liang et al. [90] claim that they only need less than 10 lines of code for their unit tests for this conversion. However, it is important to note that this conversion is done by a software tester and no automation is mentioned for this method. This concept comes up again in their later work [91], which proposes fuzzing using DeepFuzzer with fuzz drivers developed by software testers. The study of Liang et al. [91] propose the use of unit tests as part of their future work; they claim that the conversion of unit tests to create a fuzz driver would be a more convenient method rather than writing their fuzz drivers.

The research study of Lioy et al. [93] also proposes a method to reuse unit tests and convert them to fuzz drivers to fuzz IWD (Inet Wireless Deamon) architectures with AFL. The focus here is to identify software and protocol level vulnerabilities. This unit test to fuzz driver conversion process is done manually by software testers using existing unit tests. Furthermore, the work done by Myllylahti [109] also shows

a method where they scrape information from the Abstract Syntax Tree (AST) regarding all the functions to identify function parameters. Using this information, unit tests are mutated to aid the fuzzing process. This mutation process does follow an automatic approach compared to the previously mentioned methods in generating a fuzz driver from a unit test.

One of the main problems that developers of FuzzBuilder had was the time consumption for the developer to understand the library API [65]. The lack of in-depth knowledge of the library API causes the products of low code coverage. Moreover, the labour-intensive nature of manually writing the fuzz driver has also been a problem as explained in Section 2.6.10. Therefore, Jang and Kim [65] proposes a method to carry out static and dynamic analysis of unit tests in projects and generates an "executable" automatically to support grey box fuzzers such as AFL. The Fuzzbuilder takes the unit test file and the Fuzzable API; the unit test file comes as an LLVM bitcode file. Inputting this information into the FuzzBuilder requires human intervention since it needs to know which target functions require fuzzing. If the unit test follows a well-known specification, it knows how to identify the test function, however, on occasions where the unit test has no convention, a developer should intervene to provide information about the target function.

Furthermore, it gives freedom to the software testers to skip certain functions if necessary; this is mainly because certain functions (functions with excessive loops) can affect the execution speed of the fuzzer negatively. It observes two conditions to be satisfied to proceed with fuzz driver generation from unit tests: unit tests implemented as functions and unit tests are independent of each other. Then FuzzBuilder processes the API and unit tests to identify relevant functions, identifies required parameter modifications, removes unnecessary test functions and stores instrumented functions in bitcode functions.

Abstract-syntax tree

The use of AST promotes the identification of the structure of methods and classes. AST is the tree representation of the source code of the computer program [110]. AST identifies target functions to develop fuzz drivers. BigFuzz [185] uses the AST along with Direct Acyclic Graph (DAG) and user-defined functions (UDF) to understand the core program structure and identify relevant target functions. Babić et al. [7] used AST to extract the core code structure; they further used this information in program slicing. The work by Myllylahti [109] also uses AST to identify functions and function calls. In all these research studies, AST acts as a means to explore the structure of the code with functions and function calls. It allows the clear identification of relationships and structure of the code.

Program slicing

Program slicing is another approach that aids in identifying target functions. It is a technique used by software developers for abstracting for software programs [175]. It starts with a subset of program behaviour and then the process of slicing reduces that same program to its minimal form with the ability to still produce its original behaviour [175]. Therefore, the slice is an independent program that meticulously represents the original program in the same domain of the specified set of behaviour.

The research study of Kiss et al. [78] uses program slicing to identify relevant functions using a plugin called FRAMA-C. FRAMA-C identifies program statements, information about reading and writes access to relevant variables, logical annotations

and functions calls and returns. The work of Kiss et al. [78] applied slicing to identify HeartBleed vulnerability. Initially, the code base contained eight functions and fifty-one lines of code, however, after the application of code slicing this was reduced down to two functions and thirty-eight lines of code with the same behaviour as the original code base. The researchers monitor function dependencies when using functions for slicing. Function dependencies are further explained in Section 2.7.3.

Similarly, Holland [58] shows the use of control flow graphs (CFG) and data flow graphs (DFG) to produce program dependence graphs. Then these graphs act as the information source to slice the program to collect relevant components of target functions. Furthermore, Fudge [7] uses program slicing in function extraction phase for fuzz driver generation. Fudge slices C++ code. Fudge slices the AST to identify appropriate statements that need extraction for fuzz driver development. A function is considered for slicing if there are one or more calls to the target library that is a parsing API. The slicing mode extract snippets of code from the target program. Then this snippet acts as the base of the fuzz driver.

Function signature monitoring

The final step of the target identification is function signature monitoring regardless of the technique followed to identify functions to fuzz. Function signature defines the inputs and outputs of methods and functions. It is also known as method signatures and type signatures. Function signature monitoring is very important since the fuzz driver should have information regarding the type of input to send for fuzzing. There are multiple ways that research studies propose the identification of function signatures.

The work from Kelly et al. [73] uses annotations by software testers to classify the correct function signature type as explained in Section 2.7.2. Ognawala et al. [118] expresses the requirement of distinguishing function parameters from pointer types and non-pointer types. They ignore functions with double or more pointers saying that it is a limitation in their framework. Their program distinguishes between pointers and their sizes to identify the input characteristics of generated fuzz drivers through an automatic procedure that involves function signature scanning.

As explained in Section 2.7.2, Fudge uses AST to identify the program structure and then relevant functions [7]. It generates multiple fuzz drivers even for each function and then requests the software developer to choose the fuzz driver that suits the most to fuzz the target program. The fuzz driver is automatically generated with the extracted function signatures, however, there is human involvement in identifying the optimum driver for the task.

The research from Myllylahti [109] identifies the function signatures through the information gathered by the AST (Section 2.7.2). This work investigates the parameter types of the function calls to determine the function type. On the other hand, the tool of RULF [69] gathers all the API signatures to identify function signatures. The work of Jiang et al. [69] uses this technique further to extract dependencies but we explain this further in Section 2.7.3.

Intelligen by Zhang et al. [183] proposes a unique strategy to identify functions and extract appropriate function signatures for fuzz driver development. Intelligen is an automatic fuzz driver generation software. It identifies functions for fuzz driver development by evaluating the function priority. It counts statements that dereference a pointer or those that call other functions for memory processing. It uses LLVM IR that contains load and store instructions regarding memory interactions of reading and writing through pointers and call instructions that call functions. This information

helps to trace back and identify the priority of any function. The Intelligen locates entry functions and proposes multiple entry functions with recommendations. The user can pick the function that they want to fuzz; so they can either let Intelligen recommend functions automatically or manually intervene and change the function selection.

Intelligen scans the function intermediate representation (IR) to search and identify comparison instructions. It would further generate more code to check if the relevant value is assigned to the argument. Furthermore, Intelligen adopts memory allocation and deallocation functions to prevent memory leaks in the fuzz driver. Intelligen uses an algorithm to sort the entry function locator. Then it picks those functions with high priorities as its entry functions for the fuzzer.

The experiments of Zhang et al. [183] claims that compared to the pattern matching in Fudge [7] and API function searching in FuzzGen [62], Intelligen can identify vulnerable functions more effectively through its adapted ranking algorithms. In the case of FuzzGen [62], the function signatures are gathered by monitoring C/C++ header files. It produces a meta file with all the functions exiting the program, and then it extracts all the type signatures from the header files and extracts their signatures. Then this information aids to synthesise the fuzz driver. The order of appearance for function calls inside the fuzz driver is dependent upon the API dependencies (Section 2.7.3).

The research study of Blair et al. [11] proposes HotFuzz, which considers every single function as an entry point without discarding functions. HotFuzz tries to fuzz every single function presented in the input by extracting function information inspired by the micro-fuzzing proposed by Godefroid [47]. Micro execution proposes a methodology to execute fragments of code without user-provided input data and a test-driver [11] by identifying the location of the functions or code in the executable (exe) file or the dynamic link library (dll). For the testing purpose, a customised run-time virtual machine (VM) starts the code execution at the target location. It catches all the memory operations and carries out memory allocation on the fly to perform read/write memory operations. Furthermore, it provides input values according to a customisable memory policy. The first VM prototype that offers this is called MicroX, which allows micro executions to be carried out in x86 binary code. This process only requires the dll or exe and it does not require the source code, input data, debug symbols and a fuzz driver. MicroX automatically identifies the input and output interfaces of the program. Hence, MicroX does not need to identify functions and function signatures and as a result, it bypasses the fuzz driver generation altogether.

2.7.3 Function dependency identification and fuzz driver synthesis

When a fuzz driver is synthesised, it is important to have an understanding of the core of the target program. If the fuzz driver is sending a stream of data to multiple functions, it should know the order in which those functions are to be called to avoid crashes occurring in the fuzz driver itself that can cause false positives.

Section 2.7.2 implicated methods which researchers followed to identify the program structure, and functions for the fuzz driver development process. The next step is to identify the flow of these functions and synthesize the fuzz driver. Hence, we explored these techniques in this section in terms of manual function dependency identification, API dependency analysis, control flow graphs, data flow graphs, project control graphs and the avoidance of function dependency.

Human interference: function dependency identification by a software tester

The software tester identifies function dependencies on the majority of occasions. As explained in Table 2.3, 50 research studies showed the writing of fuzz drivers manually. Similarly, 52 research studies show that software testers manually identify function dependencies. This process interconnects with Section 2.7.2 expressing the human involvement in fuzz driver identification.

Function dependency identification requires a good understanding of the target program and the programming language [6, 33, 129]. Function dependencies become important when the fuzz driver targets multiple functions [62, 183]. However, if the requirement is to build a fuzz driver for a single function, function dependencies are irrelevant. We further explore such occurrences in Section 2.7.3. Our results show that researchers follow programming analysis techniques to recognise function dependencies.

Program analysis

Program analysis involves the monitoring of the control and the data flow of the program. The information it extracts helps to maintain reliability, correctness and security. Control flow monitors the order of statements in the program while the data flow monitors the correctness of data value in its execution [4]. The control flow of the program is the workflow of the methods that the program executes in an orderly manner. It is the flow of operations or tasks in the program.

The data flow of a program is the flow of data from the source to the destination. This includes information regarding data transformations in this process. Both the control flow and data flow are directly related to each other since they both drive the program execution. A common method followed by researchers to model these two concepts is through the use of CFG and DFG. For example, the research technique adapted by Allen [4] proposes the use of CFG, which gives information regarding program paths and their execution information.

Our studies show that multiple research studies use CFG and DFG for programming analysis. This includes both automatic and semi-automatic methods that develop fuzz drivers. The study done by Kiss et al. [78] explains the importance of identifying the functions with taintable data flow when observing vulnerabilities through data flow analysis. BugFuzz by Zhang et al. [185] shows the combined use of the control flow analysis and data flow analysis. They extract the behaviour of the data flow of user-defined functions (UDF) in the program while monitoring its control flow coverage. They name this process joint data flow and user-defined function coverage (JDU coverage). They use JDU information to identify dependencies and aid the fuzz driver development process.

As it shows in Section 2.7.2, Fudge uses a slicer to analyse AST to determine appropriate functions for fuzz driver generation. The next step of its fuzz driver synthesis process is to identify function dependencies; it fulfils this task by collecting dependant statements through a control flow graph [7]. Fudge uses an algorithm on a given AST function to explore both forward and backward dependencies. If it contains a function with a library call that requires fuzzing, then it collects all the calls of that particular library as seed statements. Once Fudge collects these seed statements, they check all their dependencies until it reaches a fixed point. Fudge explores these transitively relevant statements using both data flow and control flow dependencies.

FuzzGen [62] collects functions that infer the API. Then it generates an Abstract API Dependence Graph (A^2DG graph) through this information. This explores the function dependencies within the code base that requires fuzzing. A^2DG extracts dependencies and interactions between API calls. Moreover, it expresses the order of function invocation in ascending order and those functions that depend on each other. A^2DG graph implicates information regarding control dependencies and data dependencies.

Control dependencies show how to call APIs and on the other hand, data dependencies show connections between arguments and return values in the API calls. This clarifies if a return value of one API call is an argument in another API call [62]. Therefore, it is evident that A^2DG graph consists of sequences of valid API calls similar to a control flow graph. The edges in the A^2DG represent the control flow between API calls while its nodes represent calls from the API function.

To build an A^2DG graph, initially, it is required to generate an A^2DG for each root function in every consumer. Secondly, these A^2DG are coalesced into a single A^2DG [62]. If the target function is a library, FuzzGen develops control flow graphs for all the exported API functions, otherwise, it takes the main function as a starting point. After the A^2DG development, nodes of the graph represent APIs. After this process, FuzzGen merges multiple A^2DG into a single A^2DG and this process involves the migration of children, sub-trees and other common nodes [62].

Furthermore, FuzzGen carries out argument flow analysis, which involves data flow dependency identification [62]. It identifies these data flow dependencies by analysing libraries for arguments and their return values across API calls through static per-function alias analysis. Secondly, it checks dependencies across functions to identify their dependability. After gathering information about the control flow and the data flow of the sliced functions, FuzzGen finally uses this information to synthesise a single fuzz driver that can fuzz all the relevant functions in the target program.

A tool called HyDiff proposes a manual process in which to develop fuzz driver [115]. However, this research study explores the topic of software analysis by identifying the flow of the program functions. To carry out these criteria, the researchers rely upon control flow graphs. In particular, Noller et al. [115] discuss the construction of inter-procedural-control-flow-graph as a part of their fuzzing component to identify function dependencies. This study focuses on this concept to aid the symbolic execution rather than aiding the fuzz driver generation; fuzz driver generation is a rather manual task in this research study.

RFuzz proposes a technique to generate fuzz drivers for AFL and the development process of these fuzz drivers is reliant on the function dependency identification through a control flow graph [81].

RULF developed by Jiang et al. [69] proposes a technique to automatically generate fuzz drivers for software written for Rust libraries. The process of developing a fuzz driver using RULF is heavily dependent upon the use of an API dependency graph similar to the A^2DG approach followed by FuzzGen. The generated fuzz drivers through RULF take the form of a sequence of API calls. They use the technique of breadth-first search of sequences of API calls under the length threshold on the graph. Then they do a backward search for each collected API due to length limitations. After this, it refines the sequence of the minimum subset that would cover the same APIs [69]. The first step to generating this API dependency graph is the extraction of all the public API signatures from the system under test.

RULF relies on an external tool called rustdoc to generate this API documentation based on Rust code base [69]. Then RULF extracts these API signatures and generates a dependency graph by defining and modifying call types according to their needs.

Using the API dependency graph, RULF generates API sequences and subsequently, this information gives the order of API calls in the fuzz driver synthesis process.

Similarly, Winnie [71] focuses upon API call monitoring, control and data flow monitoring when it comes to function dependency analysis. After Winnie identifies relevant functions to develop fuzz drivers, it starts the process of reproducing a set of API sequences that would reach the required functions that need fuzzing. They call this the harness skeleton. It includes function calls that connect to the system under test, function prototypes identified through static analysis and auxiliary code such as main, helper functions and forward declaration. to propagate the fuzz driver.

Furthermore, it identifies situations where there are multiple threads and only keeps threads that trigger file-related APIs and remove others to maintain the fuzz driver correctness [71]. Then Winnie explores both control and data flow dependencies to make sure the fuzz driver is built within the correct program logic. It uses static analysis for this task and observes the control flow within API call paths in terms of invoking the function, returns values and termination. However, the current product cannot analyse complex flows that contain multiple variable operands and assignments inside conditional statements. Winnie relies upon human intervention to sort out such situations. Winnie also processes data flow dependencies to identify relationships that are between arguments and return values in functions. Winnie checks flow from return values, argument retrieval from memory using pointers and repeated uses of the variables on these occasions and map all the data dependencies. Then using this information, Winnie generates the order of logic of API sequences in the fuzz driver for its synthesis.

The work proposed by Holland [58] uses a platform called Mockingbird to aid the fuzz driver generation process. Similar to other research studies, they also rely on control flow graphs and data flow graphs to understand the function dependencies. Mockingbird framework initially constructs the control flow graph and the data flow graph of the target program. They generate a control flow graph with nodes and edges that represents program statements and transitions of control flow. The behaviours of functions represent paths from roots to their leaf. Similarly, a data flow graph is in use to model the data relationships in the program. It identifies dependencies by checking operator nodes, variables, primitive values and other assignments.

Furthermore, Holland [58] proposes the analysis of inter-procedural control flow through the use of a call graph. Call graphs represent inter and intra-procedural nesting structures of loops and how they climb up the chain of commands [58]. Furthermore, they propose a methodology called a projected control graph for further analysis of the program and for the simplification of the control flow graph with a control dependency graph which shows the dependencies of statements within the program. A projected control graph is a simplified version of the control flow graph that only contains branches from a set of events of interest along with the control dependent branches [156].

Above mentioned processes implicate how program analysis aids the fuzz driver synthesis by recognising the order of function calls. Our results show that 15 studies use control flow graphs, data flow graphs, project control graphs and API dependency graphs to recognise program logic for fuzz driver synthesis. However, on some occasions, the function dependency is irrelevant for fuzz driver synthesis.

Avoidance of function dependency identification

According to our results, seven research studies [11, 38, 47, 65, 73, 84, 183] show that there is no requirement to identify function dependencies when developing fuzz

drivers. It is important to note that they are all combinations of manual, automatic and semi-automatic fuzz driver generation methodologies and they express this idea due to a variety of circumstances. The work by Kelly et al. [73] only produces a fuzz driver per one function at a time. If the user annotates multiple functions, it picks the function which has the last annotation and generates a fuzz driver for that particular function. The purpose of this tool is to create a fuzz driver for a single function that the user wants to fuzz rather than fuzzing the whole codebase at once. Therefore, function dependency identification is not a requirement for this study.

FuzzBuilder [65] builds fuzz drivers by converting existing unit tests into fuzz drivers. Therefore, Fuzzbuilder takes information regarding function dependencies from unit tests and build the fuzz driver accordingly. The identification of function dependencies is not a priority in studies that promote the reuse of fuzz drivers.

The research done by Fioraldi [38] reuses an OSS-Fuzz fuzz driver, which does not require the function dependency analysis. Similarly, the work done by Le [84] shows the reuse of the same LibFuzzer fuzz driver for the fuzzing engine Kluzzer for the same code base, thus no requirement to modify function dependency. Intelligen uses a special algorithm as explained in Section 2.7.2 and identifies the function prioritisation to synthesise the fuzz driver accordingly [183]. Micro execution proposed through MicroX by Godefroid [47] follows a method where all the functions are fuzzed without the use of a fuzz driver by directly invoking the functions. It is later adopted in the work by Blair et al. [11] and the function dependency analysis is not mentioned in both of these studies.

2.7.4 Interfaces fuzzed

Our identified 102 research studies focus on fuzzing a variety of software systems using fuzz drivers. We categorised these fuzzed interfaces in terms of the programming language. It is important to note that some fuzzers and fuzz drivers can fuzz more than one programming language. For example, a fuzz driver created for LibFuzzer can fuzz programs written in both C and C++ with minor modifications to the fuzz driver. This is explained in research studies such as Huang et al. [60], Fioraldi [38], Heelan [56], Jang and Kim [65] and they show some of the prime examples for this paradigm.

Our results show that from 102 papers, the majority of research studies fuzzed C interfaces (65 research studies) using a fuzz driver. Secondly, 34 studies focused on C++ programs. It is evident that 22 research studies focused on both of these languages implicating the high amount of research done for those programming languages. Comparatively, the third-highest focus is on Java with 15 studies. Four studies [57, 70, 71, 96] focused on fuzzing C#/.NET interfaces. Out of these four studies, two studies by Jung et al. [71] and Jung [70] both talked about the same study: Winnie. This is because the research paper publication [71] is inspired by Jung's thesis [70].

Another four studies [36, 96, 97, 109] focused on fuzzing Python programs while three studies [34, 48, 96] focused on fuzzing JavaScript programs. Two studies [96, 134] fuzzed programs written in Ruby while languages such as Rust [69], Haskell [96], Perl [96] and OCaml [134] only appeared once. This implicates the distribution of interfaces fuzzed by multiple fuzzers in this research domain. More focus is given to C and C++ fuzzing with fuzz drivers. Java receives a moderate amount of focus but we do believe that there is certainly room for improvement (Section 2.11). Minor attention is given to technologies such as .Net, Python and JavaScript, but they require further research studies to explore more interesting and potential topics in this area.

2.7.5 The use of fuzzers

Our research study shows the use of multiple fuzzers across this domain for fuzzing (Table 2.4). The generation of fuzz drivers has to accommodate the conditions of these fuzzers. There are some studies that use multiple fuzzers for evaluation [17, 65, 75, 116, 158]. It is evident that 51 research studies propose their own fuzzer along with fuzz driver/s to fuzz the target program. On the other hand, 51 studies propose the use of LibFuzzer and AFL (or a variation of AFL such as AFL++, AFLFast and AFLNET). In particular, 29 research studies propose AFL or an AFL variation while 22 research studies propose the use of LibFuzzer. It implicates the popularity of AFL and LibFuzzer as fuzzers since half of the research studies used this software. The high focus on these two software is mainly due to their offer of standardised structure for fuzz drivers and fuzzing. Furthermore, LibFuzzer and AFL are widely used for automated or semi-automated attempts at fuzz driver generation while using C and C++ as programming languages. This explains the high number of fuzzing interfaces written in those two languages compared to others as explained in Section 2.7.4.

Table 2.4. Types of fuzzers.

Fuzzer	Count
AFL (or one of its variations)	29
LibFuzzer	22
Other	51

2.7.6 Availability of source code for research replication

It is a recommended practice to make the source code used for experimentation publicly available since it allows other researchers to replicate the experiments. From 102 research papers, only 30 papers made their source code open source. According to our findings, the most commonly used languages to develop software are C, C++, Java and Python. As explained in Section 2.5, the topic of fuzz drivers shows varying in-depth in each research paper. This means that while some papers solely focus on fuzz driver development, other papers only mildly mention the existence of a fuzz driver. Their research is written in the domain of fuzzing with supporting open-source code but on occasions, that code is not relevant to fuzz drivers nor fuzz driver development. As a result, only eight research studies have source code available that is related to fuzz driver development from these 30 studies. Some research studies depict fuzz drivers in the main text of their research paper. Out of 102 papers, 25 papers showcase a fuzz driver in the main text while explaining its usage. Table 2.5 shows the summary of these results.

Table 2.5. Open-Source access to findings.

Availability	Count
Open-source code available	30
Open-source code is built for (full/partial) automatic fuzz driver generation	8
Showcase of a fuzz driver in the text	25

2.8 Evaluation methods

There are multiple methods used by researchers to evaluate the fuzzing campaigns that use a fuzz driver. We will first observe the most common occurrence of evaluation methods in research papers related to fuzzing and then move on to fuzz driver evaluation. The most commonly used evaluation criteria in these research papers are the effectiveness of the fuzzer. Our results show that the identification of a number of bugs is the most common fuzzer evaluation method followed by code coverage and the duration of the fuzzing. It is important to note that these papers do not stick to one evaluation method; they try to evaluate their target functions with a variety of metrics to validate their results.

The speed of the fuzzing such as executions per second was also measured as a metric for evaluation; furthermore, certain projects purely focused upon identifying a known bug (e.g. Heartbleed Bug) [78] rather than trying to identify an array of bugs. When they identify a known bug, they further evaluate these bug identification process through other methods such as the duration for the bug identification, code coverage before the bug identification or the number of inputs before the crash. Then this information is used to enhance the fuzzing process by improving those value metrics to identify that particular bug in a quicker time, less coverage or with less seed input. Table 2.6 shows the summary of these results.

TABLE 2.6. Fuzzer evaluation methods.

Method	Count
Number of Bugs	68
Code Coverage	49
Fuzzing time	22
Fuzzing speed	12
identification of a certain bug	7
Number of inputs per crash	1
Not Stated	23

Those research studies that purely focus on fuzz driver development introduce further evaluation metrics while following above mentioned methods. On occasions where the fuzz driver development is automated, those studies compare automatically developed fuzz drivers with manually developed fuzz drivers [7, 62, 73, 183]. Furthermore, on some occasions, they carry out comparisons with fuzz drivers that are built automatically from a similar automatic fuzz driver generation software [183]. On these occasions, they try to compare the claims of their effectiveness against each other by measuring factors such as the number of bugs, code coverage and fuzzing speed. Furthermore, there is an identification of other metrics such as the number of APIs, interfaces and interface models [65] to solidify the claims of the product effectiveness.

It is clear that metrics explained in Table 2.6 are the most commonly used metrics for evaluating the fuzzer performance according to our findings. Those studies that focused on automatic/semi-automatic fuzz driver development also proposed these metrics in their studies. However, in addition, they further proposed metrics explained in Table 2.7 to compare fuzz driver effectiveness.

TABLE 2.7. Fuzz driver evaluation methods.

Method	Count
Comparison with manually built fuzz drivers	4
Interface model evaluation	1
Number of interfaces/API functions identified	1
Comparison with other automatic fuzz driver generators	1

2.9 Fuzz driver development software that does not associate with a research study

When we designed this study and developed the attribute matrix, we focused thoroughly on research studies rather than software products. However, it has come to our attention that there are software tools that could also aid the fuzz driver development process but lack research studies. These tools did not appear as results of our search strategy on Google Scholar (Section 3.3), however, since we intend to cover all the aspects of fuzz driver development in this research study to satisfy the first research question, we mention these tools in this section. It is important to note that these studies are not associated with our synthesis of results (Section 3.3) since they do not satisfy the search strategy (Section 3.3) and paper filtration criteria (Section 3.3).

Code Intelligence is a software security company with the primary goal of providing fuzzing solutions for software. They provide two products: Jazzer and CI Fuzz [28]. Jazzer is a coverage guided fuzzer developed to fuzz Java Virtual Machine (JVM) platform and it is based on LibFuzzer [67]. It is a Java fuzzer and it requires a fuzz driver to reach the target program. Their Github repository documentation provides information regarding step by step processes to develop a fuzz driver [67]. On this occasion, a software developer needs to write the fuzz driver.

Code Intelligence [28] also has a product called CI Fuzz, which is focusing on fuzzing multiple languages. This is not a released product yet, however, their demo material shows its capability to automatically generate a fuzz driver when a software developer selects a particular function [27]. This is somewhat similar to the annotation-based fuzz driver generated proposed by Kelly et al. [73], however, CI Fuzz seems to come with their own graphical user interface for function selection according to their demo [27]. As of this moment, they only issued a demo video and a brief description of this product and no code or further information is available.

Microsoft proposes their fuzzing tool called OneFuzz [105] that uses LibFuzzer `LLVMFuzzerTestOneInput` function prototype for fuzz driver development. The developer has to manually write the driver program for the fuzzing process compared to the automated method proposed in Winnie [71] for Microsoft products. This product is available through GitHub as an open-source tool to be integrated with Microsoft products for fuzzing [105].

Bogenberger [14] proposes a concept to automatically generate fuzz drivers for code written in C language. Similar to FuzzGen [62], they carry out automatic parsing and analysis of the header file of the library to identify functions and their signatures. Using this information, their tool generates a fuzz driver to fuzz C programs. They measure its performance by comparing the coverage of the automatically generated fuzz drivers with handwritten fuzz drivers from a developer for the same code base. They claim that false positives among the automatically generated fuzz drivers were high compared to hand-written fuzz drivers and they had to face limitations on the

amount of fuzzable libraries due to relying only upon C specific features and using header files to extract function information [14]. This product is not available open-source nor its research findings are available publicly for further evaluation.

2.10 Research gaps and future directions

We derived future directions by thoroughly analysing the results explained in the above sections. In particular, we analysed future directions of research papers and compared the techniques that each of the studies proposed to identify gaps in research and potential future research studies. A summary of a few future directions are available in this Section, however, these methods are thoroughly explained in Chapter 4.

Our results show that fuzz driver generation research is highly focused on C, C++ and Java. Hence, further research studies should be carried out to expand fuzz driver research to other programming languages. From multiple research studies [65, 109], we identified that unit test to fuzz driver conversion is a manual task. Thus this conversion process could be automated. The function importance ranking technique proposed by Zhang et al. [183] could be expanded to programming languages other than C and C++. The work of Godefroid [47] proposes Micro fuzzing as an alternate strategy to fuzz driver development. The efficiency of these two methodologies could be analysed to find the most effective vulnerable detection strategy. Furthermore, when identifying information about the target function, observation on documentation files and existing comments would help recognise function types and their features more effectively. More details about these future directions with explanations are available in Chapter 4.

2.11 Limitations and threats to validity

We used Google Scholar [136] to extract results for this study rather than extracting results from research libraries such as dblp, IEEE, and ACM Digital Library. Our goal is to explore all the content related to fuzz driver development in this study. Therefore, if we were to explore those libraries, we must search multiple libraries for a complete set of results. All of these libraries have multiple search algorithms hence the same search string will not return consistent results. Since Google Scholar indexes a vast amount of research papers from multiple libraries, we used it for the paper search by keeping the consistency of the search results.

We did not consider the ranking of the conferences when extracting papers. Therefore, we extracted all the research materials written in this domain for analysis without discarding research materials based on ranking levels. Also, we gathered sources such as undergraduate, Masters and PhD thesis written in this domain, which is otherwise not captured if we followed a certain ranking based resource extraction. However, to maintain the quality and the relevancy of this study, we carried out our own quality assessment criteria as described in Section 3.3.

Since we used Google Scholar [136] as our resource base, there are software products with no research studies that are not identified through our methodology. However, we did extract information regarding software tools that aid fuzz driver development through Google and GitHub by systematically searching identified keywords identified in Section 3.3 and included that information in Section 2.9. It is important to note that since we collected these sources outside of our research methodology (Section 3.3) and since they do not have an attached research study, we did not include

them in the attribute matrix. However, their existence is explained in this study to answer our first research question (Section 3.3).

2.12 Summary

In this chapter, we report on our systematic literature review on fuzz drivers that aid fuzzing. We designed an attribute matrix by categorising the role of fuzz drivers in the fuzzing life cycle into multiple major areas. Based on a thorough analysis of 102 research studies, we summarise,

- The fuzz driver development life-cycle consists of target identification, program analysis and driver synthesis. When a fuzz driver is written manually, both function identification and dependency analysis are done by the developer. However, when the fuzz driver development is automatic or semi-automatic, researchers use methods such as annotations, AST extractions, header file monitoring and program slicing to identify functions. They further use control flow graphs, data flow graphs, data dependency graphs and project control graphs for function dependency identification. These methods provide the order sequence of API calls for fuzz driver synthesis.

- Nearly half of the research studies showcase the manual writing of fuzz drivers. However, there are novel automated and semi-automated methodologies proposed to generate fuzz drivers published from 2013 onwards.

- The most commonly demonstrated fuzz driver evaluation techniques are the comparisons of manually and automatically generated fuzz drivers, interface model evaluations, the measure of the number of APIs and the comparison with other automatic fuzz driver generators.

- The most commonly explained metrics for fuzzer evaluation are the number of bugs, code coverage, fuzzing time, fuzzing speed, identification of a certain bug and number of inputs per crash.

- When it comes to fuzz driver development, the most commonly explained aspects of exploration by the authors of the surveyed articles are the methods of reaching a bug, types of bugs, the effect of seed corpus on a fuzz driver, the effect of multi-threading, speed, timeouts, difficulties of manual fuzz driver development and the requirement for automation of fuzz driver development process.

- The majority of research studies (78%) showed fuzz driver development as a minor activity in research studies. It is important to note that these studies proposed fuzz driver development as a manual task done by a software developer. 7% of papers proposed novel approaches in developing fuzz drivers as the main contribution while further 8% proposed novel methods while not being the main contribution of the paper. 6% of papers showed extensions from previous novel work while 2% of papers proposed new techniques to bypass the fuzz driver generation process from the fuzzing life-cycle.

- C, C++ and Java are the most commonly used languages in fuzz driver generation.

Overall, we explored the importance of fuzz drivers in the domain of fuzzing while analysing its development methods and the life cycle. We mapped programming

languages used for fuzz driver development with available source code and we analysed potential evaluation criteria in terms of fuzzing and the role that fuzz drivers play in the domain of fuzzing. Therefore, we successfully answered our first research question by carrying out this empirical research study.

Chapter 3

What is the best fuzz driver generation strategy?

3.1 Introduction

There are multiple ways of developing a fuzz driver as we identified in Section 2.4.3. These identified techniques for fuzz driver development are manually writing a fuzz driver, fuzz driver generation semi-automatically with human-in-the-loop approaches, fully automated fuzz driver generation techniques and fuzz driver existing as a built-in-component in a software product.

Our results from Table 2.3 show that manually writing a fuzz driver is the most common practice among researchers and software testers. However, Figure 2.3 depicts that post-2018, there is high interest in developing automated and semi-automated methods to generate fuzz drivers. Thus, in this chapter, our first motivation is to identify the best semi-automated and automated fuzz driver developing tools that are available as open-source products and compare their effectiveness against manually written fuzz drivers. We did not analyse the fourth technique where fuzz driver exists as a built-in component as mentioned in Section 2.4.3 due to the lack of open-source tools for results replication.

The results from Table 2.7 show that there are studies that compare different fuzz driver development tools against each other, which follow the same fuzz driver development strategy. There are four attempts [7, 62, 73, 183] taken to compare two fuzz driver development techniques (manually developed fuzz drivers compared with one other technique). However, according to the findings in Chapter 2, no study has been carried out since 1990 to compare fuzz driver generation techniques from multiple open-source products to identify the best state of the art fuzz driver development strategy. Therefore, we explore this research gap in this study to answer the second research question.

Our results from Section 2.7.4 show that C is the most common programming language in fuzzing research, hence we focus this study on C programming language fuzzing. Moreover, we identified that LibFuzzer [94] is a widely used C programming language fuzzer (Section 2.7.5), hence we use this open-source tool as the fuzzer in our case study. As shown in Table 2.6, identification of the number of bugs and code coverage are the most popular metrics in recognising the effectiveness of the fuzzing campaign; thus they are our evaluation metrics. Furthermore, we analyse the development strategies of multiple fuzz driver generation tools and explore potential improvements for their current standards. Thus, answering the second research question.

There is no mention of any external factors that could affect the fuzz driver performance in Section 2.8. From the majority of the studies we explored, it is clear that their conclusions focus thoroughly on the performance metrics of the fuzzer such as

the number of bugs, code coverage, fuzzing time, and fuzzing speed. Code complexity analysis exists in the software testing domain to recognise the maintainability and the efficiency of the code base [145]. Thus, our second motivation is to explore the effect of the varying code complexity in the target code on fuzz drivers and the fuzzing campaign.

The inspiration for this idea comes from research studies carried out in another software quality assurance domain called unit testing. Unit testing is a testing methodology where testers have to write a function to mimic the target function and verify whether the program component works accordingly with expected input [133]. The software tester has to put their manual effort and write this function for a target program function similar to writing a fuzz driver for fuzzing. There are multiple studies carried out to check the correlation between the complexity of the source code and the effectiveness of unit tests [2, 8, 141].

Code quality metrics indicate information regarding design flaws in the software code [124]. There are studies done involving code quality metrics and fuzzing [21, 61, 146]. However, such a study has never been carried out to identify the effect of fuzzing along with fuzz drivers. Thus, opening up a research gap to explore. As a result, we use source code quality metrics to explore, how the complexity of the target code compares to the metrics gathered through fuzzing (code coverage) for fuzz drivers developed through multiple techniques. Our intention is to identify what requirements are needed in a fuzz driver to fuzz highly complex code efficiently. Through this experimentation, we intend to answer the third research question.

3.2 Background and related work

The Fuzz driver contains two parameters. The first parameter carries the buffer address by storing a random value generated by the fuzzer [65]. The second parameter stores information regarding the size of the buffer address. It is the task of the tester to write the body of this function according to the functionality of the target program. There are characteristics that a fuzz driver should possess to be a LibFuzzer fuzz driver [94]. When using LibFuzzer, software testers should create the `LLVMFuzzerTestOneInput` to act as the interface between the target library and the fuzzer. Hence, it is the fuzz driver for the LibFuzzer to fuzz the corresponding target code. One of the issues with manual fuzz driver development is that it is a time-consuming process as we identified in Section 2.6.10. However, there were recent attempts taken to automate this process as explained in Section 2.4.3.

There are three steps for fuzz driver development as we identified in Section 2.7.1. They are target identification, program analysis and fuzz driver synthesis. There are different levels of human involvement in semi-automated fuzz driver development approaches. Information that we gathered from Chapter 2, shows that semi-automated methods contain human involvement either in the target identification process or in the fuzz driver synthesis process (Figure 3.1). For example, Fudge [7] requires software tester involvement after the synthesis of the fuzz driver to identify the most effective fuzz driver generated from Fudge. This is because Fudge generates multiple fuzz drivers for the same function.

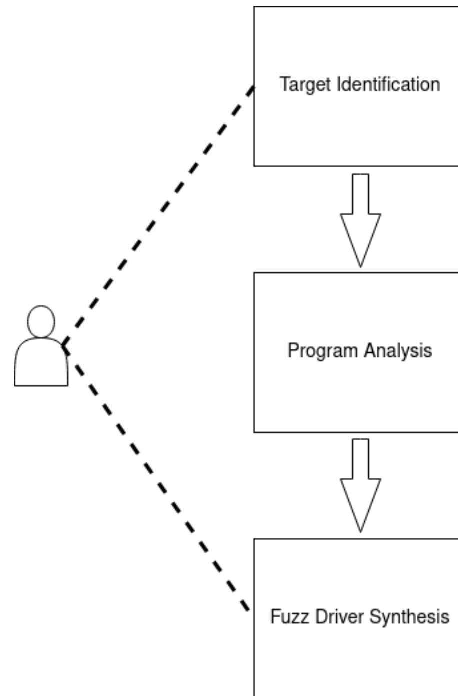

FIGURE 3.1. Human involvement in semi-automated fuzz driver development.

For this research study, our focus is on human involvement at the target function identification step because this process involves the identification of functions, function signatures and function features for a good quality fuzz driver generation as explained in Section 2.7.2. The function signature is the definition of the input and output of a method or a function. The first step in the fuzz driver development life cycle is more important to develop an effective fuzz driver compared to the process of figuring out which fuzz driver is best after it has already been generated according to our findings in Chapter 2,

From all the studies we analysed in Chapter 2, Fuzz Target Generator (FTG) [73] is the only semi-automated fuzz driver generation method that aids the first step of the fuzz driver development life-cycle. Therefore, we decided to use FTG for fuzz driver development in this research study as our semi-automatic approach. FTG is further explained in Section 3.2.1.

Multiple research studies [62, 183] propose techniques to generate fuzz drivers automatically. However, from all these approaches, only FuzzGen [62] is available as an open-source product for researchers and software testers to generate fuzz drivers automatically for target code. The authors, Ispoglou et al. [62] claim that FuzzGen can surpass the performance of manually developed fuzz drivers. Therefore, we decided to use FuzzGen as the fully automated approach for fuzz driver development in this research study. FuzzGen is further explained in Section 3.2.2.

3.2.1 Fuzz Target Generator (FTG)

FTG is a semi-automatic fuzz driver generation method proposed by Kelly et al. [73] that aids fuzz driver generation for LibFuzzer. As explained in Section 2.6.10, manually writing a fuzz driver is time-consuming and requires a lot of domain knowledge. However, FTG proposes a methodology to produce the fuzz driver directly with minor

inputs from the software tester. FTG is a software tool that is specifically designed for C.

The software tester input is in the form of annotations that are pre-defined code comments. These code comments are `@fuzztest`, `Array (array-ptr, array-len)`, `Value (parameter, value)`, `Output (parameter)`, `Cleanup (condition, function, [ params])`. This process was thoroughly explained in Section 2.7.2.

Firstly, the software tester has to identify which target function they want to fuzz. Then the tester should comment on top of that function using the appropriate pre-defined annotation to scan the target function. FTG reads the annotation, recognises the function signature, identifies relevant parameters and generates a fuzz driver that is compatible with LibFuzzer. Figure 3.2 shows this process diagrammatically.

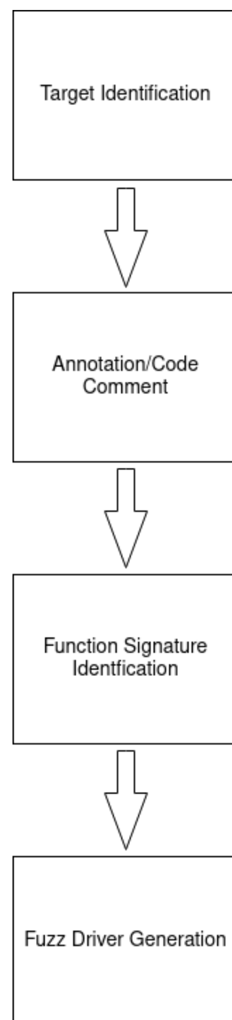

FIGURE 3.2. Fuzz driver development through FTG.

The research study of Kelly et al. [73] compares the effectiveness of the semi-automatic fuzz driver generation by comparing the fuzzing results of FTG against manually written fuzz drivers. The results show that manually written fuzz drivers still outperform FTG fuzz drivers in terms of identifying the number of bugs and producing fewer false positives. FTG is tested and evaluated by collaborating with an organisation called Cohda Wireless to identify its effectiveness [73]. Their tested

target code is not open-source, hence their results cannot be replicated. However, we will carry out experiments on FTG in open-source software products.

Furthermore, Kelly et al. [73] do not measure the code coverage in their experiments to show the effectiveness of the fuzz driver in terms of how many code blocks or branches it can cover in a given period. As we identified in Section 2.8, code coverage is an important metric in validating the effectiveness of the fuzzing campaign. Hence, we focus on measuring both code coverage and the number of bugs in our research study to compare the effectiveness of FTG with automatic fuzz driver generators (FuzzGen [62]) and manually written fuzz drivers.

The following Listing 3.1 shows an example of a simple C function. On this occasion, the target function consists of three integers; it swaps them around. It contains the `@fuzztest` comment, which informs FTG to generate a fuzz driver for this particular function.

Listing 3.1. Target C function.

```c
//@fuzztest
void swap(int *a, int *b, int *c)
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = *c;
    *c = *a;
    *c = tmp;
}
```

The following Listing 3.2 shows the fuzz driver for Listing 3.1 generated through FTG. This function accepts two parameters to fuzz the target library. The first parameter receives a buffer with random data generated by the fuzzer. The second parameter is the size of the data in the buffer. This fuzz driver uses `memcpy` to copy characters of the size of the input integers to the address. Then these values are sent to the `swap` function for fuzzing.

On this occasion, `@fuzztest` comment on the target function initiates FTG, where it reads the function signature of the `swap` function and identified three integer pointers. Using this information, an appropriate fuzz driver is generated as depicted in Listing 3.2.

LISTING 3.2. Example FTG fuzz driver.

```
int LLVMFuzzerTestOneInput(uint8_t *fuzz_input_data, size_t
    fuzz_data_size){

    size_t fuzzer_input_min_size = sizeof(int) + sizeof(int) +
        sizeof(int);

    if(fuzz_data_size < fuzzer_input_min_size) return 0;
    uint8_t * fuzz_ptr = fuzz_input_data;

    int a;
    int b;
    int c;

    memcpy(&a, fuzz_ptr, sizeof(int));
    fuzz_ptr += sizeof(int);
    memcpy(&b, fuzz_ptr, sizeof(int));
    fuzz_ptr += sizeof(int);
    memcpy(&c, fuzz_ptr, sizeof(int));
    fuzz_ptr += sizeof(int);
    (void)swap(a, b, c);
    return 0;
}
```

3.2.2 FuzzGen

FuzzGen allows the development of fully automatic fuzz drivers for LibFuzzer [62]. Application programming interface (API) is a software intermediary with a set of functions, protocols and definitions that allows the development of software that can access features of other services, applications or operating systems [117]. FuzzGen collects functions from the target program that are part of the API.

An API dependency graph is a representation of all the required dependencies of function interactions using the API [184]. After gathering this API information, FuzzGen generates an Abstract API Dependence Graph (A^2DG graph) to capture all API interactions from the target program. This process was thoroughly explained in Section 2.7.3.

FuzzGen carries out argument flow analysis, which involves data flow dependency identification [62]. FuzzGen generates a file (called meta file) with information regarding functions that include arguments and their return values across API calls. Listing 3.3 shows the meta file for Listing 3.1.

Listing 3.3. Meta file.

```
# ================ #1 FUNCTIONS ================ #
@functionhdrs
swap  external/cfunc/swap/swap.h

# ================ #1 PARAMETERS ================ #
@params
swap   a  b  c

# ================ #1 SIGNED PARAMETERS ================ #
@signedparams
swap   $RETVAL$  a  b  c

# ================ #1 FUNCTIONS ================ #
@functionhdrs
swap  external/cfunc/swap/swap.h

# ================ #1 PARAMETERS ================ #
@params
swap   a  b  c  a  b  c

# ================ #1 INCLUDES ================ #
@includedeps
external/cfunc/swap/swap.c   swap.h

# ================ #2 SIGNED PARAMETERS ================ #
@signedparams
main   $RETVAL$
swap   $RETVAL$  a  b  c
```

Low-Level Virtual Machine (LLVM) is the compiler and LLVM IR is the assembly format of the source code in the form of independent intermediate representation (IR) [95]. FuzzGen takes input in an assembly format called LLVM IR assembly. Therefore, the C files should be converted to LLVM IR files. If there are multiple LLVM IR files, they are linked up to generate one LLVM IR file input. Then the LLVM IR file is used as the input along with the meta file for the generation of fuzz drivers.

After gathering information about the control flow and the data flow of the functions, FuzzGen finally uses this information to synthesise a single fuzz driver that can fuzz all the relevant functions in the target program. Figure 3.3 shows a simplified version of FuzzGen fuzz driver development process diagrammatically.

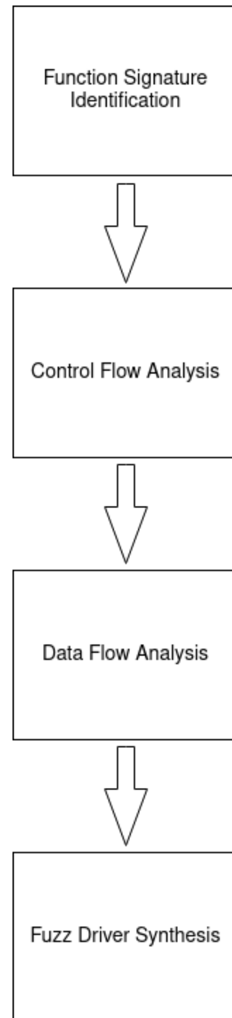

Figure 3.3. Fuzz driver development through FuzzGen.

Ispoglou et al. [62] evaluates fuzz drivers developed from FuzzGen with manually developed fuzz drivers. They use Debian and Android Open-Source Projects (AOSP) as target programs and identify 17 new bugs that weren't identified through manually developed fuzz drivers. Furthermore, Ispoglou et al. [62] shows that FuzzGen fuzz drivers can surpass the code coverage of manually developed fuzz drivers by 6.94%. They made FuzzGen source code available open-source for results replication purposes, hence we intend on testing the validity of their results in our experiments.

Listing 3.4 shows the FuzzGen fuzz driver for Listing 3.1. Similar to Listing 3.2, it identifies three inputs and sends data to the target function. However, on this occasion, it calls a method called `EatData`. The purpose of `EatData` is to make modifications to the random input generated by the fuzzer to improve the code coverage. Ispoglou et al. [62] claims that the beginning sections of the random input from the fuzzer have a high likelihood of being a valid frame or a buffer that would cause high coverage. Therefore, this frame will be destroyed unnecessarily if it is used for path selection or variable fuzzing. To avoid this, FuzzGen splits the incoming input from the fuzzer. If it is a buffer, it eats bytes from the beginning of the random input, otherwise, it eats from the end. Following this technique, FuzzGen manages to preserve the corpus and useful frames to achieve higher coverage.

LISTING 3.4. Example FuzzGen fuzz driver.

```
int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    if (size < 13 || size > 1037) return 0;

    EatData E(data, size, ninp);

    buflen = (size - ninp) / nbufs - 1;

        perm = kperm(2, E.eatIntBw( NBYTES_FOR_FACTORIAL(2) ));

        // initializing argument 'a_bSY'
        int32_t a_bSY_0 = E.eat4();
        int32_t *a_bSY_1 = &a_bSY_0;

        // initializing argument 'b_XBn'
        int32_t b_XBn_0 = E.eat4();
        int32_t *b_XBn_1 = &b_XBn_0;

        // initializing argument 'c_hBZ'
        int32_t c_hBZ_0 = E.eat4();
        int32_t *c_hBZ_1 = &c_hBZ_0;

        for (int i=0; i<2; ++i) {
            if (0) { }
            else if (perm[i] == 0) {
                swap(a_bSY_1, b_XBn_1, c_hBZ_1);
            }

        }
    return 0;
}
```

3.2.3 Other fuzz driver generators

Fudge is another automatic fuzz driver generation method with minor human-in-the-loop involvement [7]. It uses an abstract syntax tree (AST) to determine relevant statements. It slices functions identified from the AST; if that function has at least one call to the target library in a form of a parsing API. Then it uses a control flow and data flow information for the target code base and collects all the dependencies of the identified function. Using this information, Fudge synthesises the fuzz driver.

Fudge creates multiple fuzz drivers for the same code base; then these fuzz drivers are presented to the software testers through an interface, where they choose an appropriate fuzz driver, thus including minor human-in-the-loop actions. Babić et al. [7] tests Fudge on open-source projects such as Leptonica [85] and OpenCV [120]. The authors claim that they identified unique bugs in these projects, however, there is no comparison of these identified bugs and their code coverage with manually developed fuzz drivers or other fuzz driver development techniques. Furthermore, its source code is not made available for results replication, hence we could not use this tool and its techniques in our experiments.

Intelligen [183] compares automatically generated fuzz drivers with their manual counterparts. The authors, Zhang et al. [183] propose Intelligen which is also an automated fuzz driver development method. It calculates vulnerability priority for all the functions by counting the number of statements that dereference pointers or call

memory processing functions. Then it takes high priority functions as entry functions compared to taking all the functions similar to FuzzGen [62].

The study from Zhang et al. [183] compares Intelligen against Fudge [7] and FuzzGen [62] using the code from open-source Android projects on Google's fuzzer-test-suite [44]. Their results show that IntelliGen outperforms FuzzGen by covering 2.03 times more code blocks and also it outperforms Fudge by covering 1.08 times more code blocks in terms of code coverage. As a result, IntelliGen identified ten more bugs compared to FuzzGen and Fudge. However, Zhang et al. [183] does not make their source code available for further studies similar to FuzzGen, hence their results cannot be replicated and we cannot use this software product in our study to validate its effectiveness.

The research by Jung et al. [71] introduces Winnie, which is another automated fuzz driver generation tool. Their results show that they outperform manually developed fuzz drivers in terms of code coverage and bug identification, however, it is developed for fuzzing Windows applications and therefore, it is out of scope for our study.

3.2.4 Fuzzer metrics for LibFuzzer

To analyse the performance of the fuzzing campaigns with fuzz drivers, we used software bugs and code coverage as metrics since they are the most popular metrics used in the fuzzing domain (Section 2.8).

A software bug is a flaw, error or fault in the computer program [52]. The purpose of fuzzing is to recognise unexpected stopping of the fuzzing campaign due to a crash in the target program and identify whether it is a result of a software bug. A "crash" identified by the fuzzer is not necessarily a bug. When a fuzzing campaign is run on a fuzzer like LibFuzzer, it can identify multiple software crashes that could be either actual bugs or false positives. A false positive software bug is a wrong indication of a software bug that should not exist in the given scenario [24]. Therefore, it is important to recognise and discard false positives from the identified software crashes from the LibFuzzer crash outputs.

Code coverage is used as a metric in this chapter to identify the depth of code reached through fuzzing. It measures the amount of executed code during the testing process [22]. This allows the software tester to know how much of the code is tested by the fuzzer and which parts of the program are not reached during the fuzzing campaign. The higher the code coverage the better the fuzzer performance.

The code coverage is measured in multiple ways such as statement coverage, line coverage, function coverage, basic block coverage, edge coverage and path coverage [22]. Statement coverage is the executed statements in the software divided by the total statements. Similarly, line coverage is the number of executed lines divided by the total number of lines in the target program. Function coverage is the number of functions called by the fuzzer during the test. Block coverage is the execution of lines of code within basic blocks (parts of the code without branches). Edge coverage is the edges in the control flow graph that are covered divided by all the edges in the target program. Path coverage is the number of paths executed by the fuzzer divided by the total number of paths in the target program.

Fuzzers use multiple techniques to measure the code coverage during the fuzzing process and they could differ compared to the nature of each fuzzer. Fuzzers use a technique called instrumentation by adding markers in code blocks to identify the code coverage. All the fuzzers have their unique ways of injecting instrumentation to the target function, however, the basic steps of carrying out these criteria are

identical. Sanitizers are used to detect certain behaviours of the compiler during its instrumentation; fuzzers use sanitizers to identify code coverage information in target programs.

LibFuzzer [94] provides block coverage of the target code. LibFuzzer inserts callback functions during the compiling process. One of these callback functions contains a coverage counter that gives information regarding the number of blocks covered during the fuzzing process [94]. When a new state transition of the program is generated through mutation of the input by reaching a new block, the coverage is incremented. Then the mutated input is added to the input queue and used as the starting point for the next fuzzing cycle. If the input does not reach a new block, that input is discarded. Thus, by analysing number of bugs and code coverage, we analysed the performance of fuzzing campaigns under different fuzz drivers in this research study.

3.2.5 Code complexity analysis

One of the key aspects that a software system should have is its good quality to satisfy its stakeholders and consumers. Software quality is the degree to which the software promotes maintainability, testability, reliability, interoperability, low complexity etc. The quality of the software is measured by measuring the complexity of the program code. There are multiple metrics that depict the complexity of the code such as cyclomatic complexity [100] and Halstead metrics [54]. When we fuzzed target programs with three different sets of fuzz drivers, we identified bugs and we measured code coverage 3.3.2. We wanted to further explore whether the complexity of the target code has any effect on the fuzzing and different types of fuzz drivers.

Cyclomatic Complexity

Cyclomatic complexity is a software complexity measurement introduced by Thomas McCabe to identify the maximum number of linearly independent paths in the program [100]. These linear paths promote the potential number of test cases. A program with low cyclomatic complexity is defined as a less complex program with greater maintainability. If the cyclomatic complexity of a program is high, it tends to be high error-prone and difficult to detect these errors. Control flow graphs (CFG) are a representation of paths in the software program. Cyclomatic complexity is measured using the help of a CFG. Using the CFG, the programmer can measure the number of edges E, the number of nodes N and the number of nodes that have exit points P. The commands or decisions in the program are nodes and the connections between those commands and decisions are edges. The cyclomatic complexity is calculated using the following equation proposed by McCabe [100].

$$CC = E - N + 2P$$

The ideal cyclomatic complexity should be less than 10 and no more than 20. If the cyclomatic complexity goes beyond 50, the program is deemed too complex, high risk and difficult to test [76, 144]. Thus, the high threshold for the complex code.

In Chapter 2 (Section 2.7), we identified that when manually writing a fuzz driver, the quality of the fuzz driver is dependent upon software tester's ability to understand the target program. We want to analyse whether semi-automated and automated methodologies provide a solution to this problem by recognising complex functions better than software testers.

Initially, we want to evaluate the correlation between the cyclomatic complexity and the code coverage resulting from three different fuzz driver types (manual,

semi-automatic and fully automatic). Then We want to evaluate how the code coverage resulting from three types of fuzz drivers will be affected when the cyclomatic complexity goes beyond high complexity levels (beyond 50).

These experiments will give us an indication of how the increase in complexity of the target code would affect the fuzzing campaigns for our three different fuzz driver development techniques. We intend to identify whether automated methods have the ability to outperform human-centric methods.

Halstead Metrics

Our intention is to recognise whether the static features of the target code have any effect on the fuzzing campaign initiated by our identified three types of fuzz drivers. Operators and operands in a program code define actions and operations to perform. They have an effect on the data flow within the function. When fuzzing a target program, it is important to have a good understanding of the actions of the operations and operands within a function.

We want to check whether our three types of fuzz drivers have the ability to identify parameters within functions that aid the data flow as the size and the complexity of the target code increase. Also, we want to check whether the number of operators and operands has any effect on the fuzzing campaign. Therefore by calculating Halstead metrics [54], we try to statically analyse the data elements in the target program that aids the data flow and try to analyse whether this would have any effect on the fuzzing campaign.

Halstead metrics were proposed by Maurice Halstead in 1977 [54] to determine the program module complexity straight from the source code through program operators and operands. These metrics statically measure the features of the code. Initially, it extracts four different measures from the code. They are the number of distinct operators ($n1$), the number of distinct operands ($n2$), the total number of operators ($N1$) and the total number of operands ($N2$). Then using these measures, it calculates, Halstead Length, Halstead Vocabulary, Halstead Volume, Halstead Difficulty Level, Halstead Program Level etc. Halstead program length (N) measures the sum of all the operators and operands in the target program.

$$N = N1 + N2$$

Halstead vocabulary size (n) is the sum of unique operators and operands.

$$n = n1 + n2$$

However, for this research study, we focus on measuring the Halstead volume of the target code. Halstead volume (V) is the size of the algorithm implementation and (V) is measured using the number of operators and operands in the target program [54]. Halstead volume is measured by multiplying the Halstead program length by the 2-base logarithm of Halstead vocabulary. The Halstead volume of a function should be between 20 and 1000 and the Halstead volume of a file should be between 100 and 8000. If they are above their upper limits, then the complexity is too high in those corresponding target functions and target files. Halstead Volume (V) is calculated using the following equation.

$$V = N * log2(n)$$

The measuring of the Halstead volume would give the size of the program based on the number of operators and operands. If the fuzz driver has the ability to identify

the full functioning of the operators and operands within the target function along with program paths, it would be able to reach deep parts of the code. Moreover, it would indicate the depth of data operations required for an ideal data flow for the target function. Hence, we observe the capacity of the fuzz driver to handle the data flow within the function.

In this study, our motivation is to compare how fuzz drivers that are developed manually, semi-automatically and fully automatically would perform for the same target functions under the same conditions. We would like to see how manually generated fuzz drivers fare in performance compared to their automated counterparts to answer the second research question of this thesis.

Our analysis of cyclomatic complexity would give us an understanding of the control flow of the target program and how its changes affect different fuzz drivers for the fuzzing campaign. Similarly, the measurement of the Halstead volume gives us an indication of how well the fuzz drivers will be able to understand the data flow of the program through its operations and operands. Hence by understanding the correlation between the target code complexity and the fuzz driver performance, we answer the third research question of this thesis.

3.3 Methodology

To carry out comparisons between the performance of fuzz drivers, we needed three sets of fuzz drivers: manually developed fuzz drivers, semi-automatically generated fuzz drivers and automatically generated fuzz drivers. GitHub is a hosting platform for source code that allows version control and collaboration with more than 190 million repositories [46]. We systematically gathered a collection of manually developed fuzz drivers through open-source projects stored in GitHub that were developed for fuzzing through LibFuzzer [94].

3.3.1 Collection of manually developed fuzz drivers

To collect C fuzz drivers, we searched the term "`int LLVMFuzzerTestOneInput(`" on the GitHub search with the C language filter added to the search filtration. "`int LLVMFuzzerTestOneInput(`" is how the fuzz driver is written in C language as depicted in Listing 3.2 and Listing 3.4. This search returned 27,328 results. For this research, we extracted results from the first 100 pages. We downloaded all the repositories from the first 100 pages where the above term appears with a fuzz driver. As a result, we ended up with 690 projects. All these projects contained a fuzz driver written by a software tester that targets a C function in the project. We compiled and ran these 690 fuzz drivers to check their functionality. This process showed that from 690 samples, only 148 fuzz drivers compiled and others gave multiple errors such as missing functions, classes, files and various other bugs in the source code. Therefore, this set of 148 results became our sample of manually developed fuzz drivers. The summary of this process is shown in Figure 3.4.

FIGURE 3.4. Systematic collection of manually developed fuzz drivers.

3.3.2 Generating fuzz drivers

Once the manually developed fuzz drivers are collected through GitHub, we automatically generated fuzz drivers for the same target functions using FTG [73] and FuzzGen [62]. When we used FTG, we first identified the function signature that we wanted to fuzz and then added a pre-defined comment as suggested by Kelly et al. [73] on top of the function. If the function signature is an array, we used the modified comment, as explained in Section 3.2.1. At the end of this process, we had 148 fuzz drivers from FTG to fuzz our collected target functions.

The next step was to develop similar fuzz drivers using FuzzGen. However, there was an issue with this process. FuzzGen is developed to fuzz the whole software product at once and not to fuzz a single function at a time. When a fuzz driver is written manually by a software tester, the tester can fuzz a single function, few functions or call multiple functions that are related to that original target function using the same fuzz driver. When a fuzz driver is generated through FTG, it focuses on fuzzing one function. However, FuzzGen considers all the functions that are mentioned in the header files, all across the project folder to generate fuzz drivers along with their function dependencies. If we build a fuzz driver using FuzzGen for the whole project, it cannot be compared with results gathered from manually developed fuzz drivers and semi-automatically generated fuzz drivers since these two techniques do not focus on fuzzing the whole project at once. Therefore, to rectify these issues, we made slight modifications to the fuzz driver generation process followed by FuzzGen.

As explained in Section 3.2.2, FuzzGen generates a meta file (Listing 3.3) with all the information in header files. Therefore, when developing the meta file, we only included header files with functions that relate to the target function. On occasions, where these functions exist in multiple header files with other functions, we created a header file with the relevant function signatures to avoid FuzzGen selecting unnecessary functions that are unrelated to our desired target function. Following this process, we generated fuzz drivers for the same set of functions as software testers

and FTG. As a result, three sets of fuzz drivers (manually developed by the software tester, fuzz drivers generated through FTG and fuzz drivers generated through FuzzGen) were created to focus on the same set of functions in the collected projects to compare their effectiveness. The summary of this process is shown in Figure 3.5.

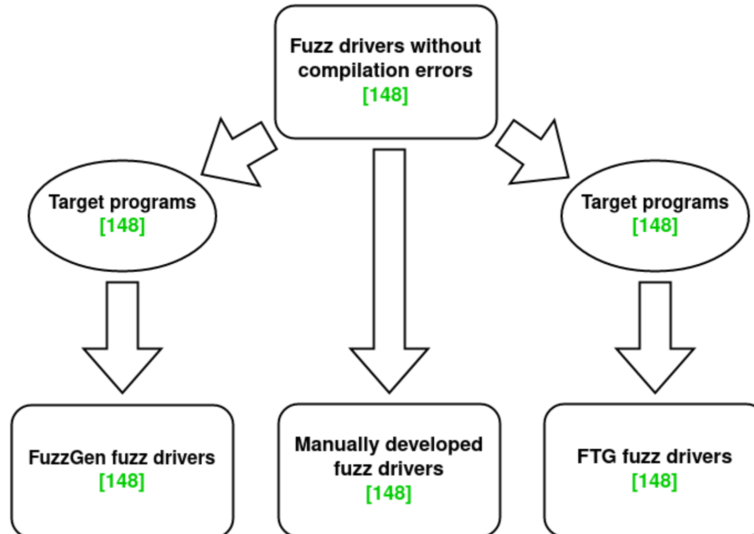

FIGURE 3.5. Three sets of fuzz drivers from gathered target functions.

3.4 Findings

In this section, we analyse how three types of fuzz driver development techniques perform in terms of identifying bugs and code coverage under the same experimental conditions. Furthermore, we analyse whether the characteristics of the target code have any effect on fuzz drivers and the fuzzing campaign. Our findings in this section help us answer the second and third research questions.

3.4.1 Fuzzing

We fuzzed each target function for an equal amount of time using the three types of fuzz drivers separately under the same conditions. We fuzzed each target function for 30 minutes. The experiments done by Ispoglou et al. [62] for FuzzGen shows that after 30 minutes the code coverage stabilises. Hence, the reason we decided the duration to be 30 minutes for our experiments. All the experiments were run on an Ubuntu 20.04 with Quad-Core Intel i7-8565U and eight GB of RAM. We built the code using Clang 9.0 with the help of AddressSanatizer [138].

We calculated the number of bugs, code coverage and false positives (if any) for each fuzz driver to compare these three fuzz driver development methods. Some open-source projects made their seed input files available for fuzzing and for those that did not offer seed input files, we manually created them for experiments. We kept the seed input consistent for all three fuzz drivers for each project by keeping it as the control variable. Therefore, the code coverage results are purely based on the effect of the fuzz driver without any external factors affecting its functionality.

Once the fuzz target fuzzes the target function for 30 minutes, we manually inspected all the identified crashes with their outputs and fuzzer input data to recognise whether each of the crashes is an actual software bug or a false positive. Once the

fuzzing campaign is run and bugs were identified, those bugs very verified by re-running the campaign two more times to ensure that the crash is replicable. We took this precaution to further validate our results.

One of the features of LibFuzzer when identifying bugs was that when LibFuzzer identifies a bug, it stops before the given time limit. Thus, the fuzzing campaign should be restarted when it stops. However, this restart re-performs all the initialisation processes. It is important to note that this is not a limitation of the fuzz driver but a limitation of the LibFuzzer. To rectify this, we ran the fuzzer in multiple parallel workers [94], so if one worker stops, others are still running for the given period.

Fuzz driver compatibility

We fuzzed all 148 target functions with three corresponding fuzz drivers with LibFuzzer [94]. As explained in Section 3.3.1, the 148 target functions that we collected are compatible with all 148 manually developed fuzz drivers. However, due to different capacities to identify function signatures, all of these 148 target functions were not compatible with FTG and FuzzGen. As shown in Table 2.1, only 95 out of 148 projects were compiled with FTG. Similarly, target functions from certain projects had problems compiling with fuzz drivers generated using FuzzGen. As a result, only 116 projects out of 148 were compiled with FuzzGen. The results in the Table 3.1 show the summary of fuzz driver compatibility with target functions.

Table 3.1. Fuzz driver compatibility with target functions.

Fuzz Driver Development Method	Compatible projects
Manually Developed Fuzz Driver	148
FTG	95
FuzzGen	116

When it comes to developing fuzz drivers using FTG and FuzzGen, their ability to function signature identification varies in comparison. Our findings show that they both identify simple parameters in the function signature such as integers and strings. However, when pointers are included in the function signature, their performance starts to change compared to each other in terms of developing valid fuzz drivers. When there is a character pointer in the function signature, both FTG and FuzzGen identify the character pointer but fail to convert the use of the character pointer correctly in the fuzz driver, thus resulting in a faulty fuzz driver. This is the same for function pointers for FTG where it fails to identify them while FuzzGen identifies and generates valid fuzz drivers for function pointers. FuzzGen and FTG seem to be compatible in recognising other function signature types such as arrays, multiple integer pointers, and void pointers. Particularly, FTG is effective with its special commenting style for function signatures for arrays while FuzzGen identifies arrays automatically without any external aid. FTG only grabs a function signature of the annotated target function without identifying any related functions and FuzzGen identifies the function signature of the target function and all its dependant functions.

3.4.2 Analysis of bug identification

The results in the Table 3.2 show that manually written fuzz drivers found the most number of crashes through LibFuzzer followed by FTG and FuzzGen. Moreover, manually written fuzz drivers identify the most number of unique bugs (29) followed

by FuzzGen (21) and FTG (5) when we subtracted false positives from the number of crashes we identified through the fuzzing campaign. This implies that manually developed fuzz drivers are the most effective in identifying bugs closely followed by the fully automated method by identifying 8 more bugs for the given period. The semi-automated method of FTG largely underperforms against their counterparts. Our results implicate that manually developed fuzz drivers find 5.8 times more bugs than FTG and FuzzGen finds 4.2 times more bugs compared to FTG.

TABLE 3.2. Number of crashes, false positives and bugs.

Fuzz Driver Development Method	Number of crashes	False positives	Number of bugs
Manually Developed Fuzz Driver	79	50	29
FTG	76	71	5
FuzzGen	33	12	21

Then we considered the results from Table 3.1, which showed the number of compatible projects for each fuzz driver. Using this data, we calculated the percentage of the bug to project ratio in Table 3.3. The information in Table 3.3 shows that manually developed fuzz drivers identify the highest percentage of unique bugs considering the number of target functions. However, it is closely followed by the fully automatic method of FuzzGen with only a 1.5% difference. Once again, the performance of the semi-automated approach of FTG is lacklustre compared to the other two methods in terms of identifying bugs and it shows approximately four times less effectiveness compared to its counterparts.

TABLE 3.3. Percentage of bugs to projects.

Fuzz driver development method	Compatible projects	Number of bugs	Percentage bugs to projects ratio
Manually Developed Fuzz Driver	148	29	19.6%
FTG	95	5	5.3%
FuzzGen	116	21	18.1%

As we identified in Chapter 2 (Section 2.6.10), manually developing a fuzz driver is a time-consuming process that requires deep knowledge about the target codebase. Our results showed that the fully automated approach of FuzzGen nearly matched the performance of manually developed fuzz drivers in terms of identifying bugs with only 1.5% less performance. Of all three methods, the fully automated method takes the least time to implement (it takes a few seconds to generate a fuzz driver through FuzzGen). Therefore, it is clear that the fully automated methodology of FuzzGen gives a higher success rate per effort compared to manually developed fuzz drivers even though manually developed fuzz drivers outperformed the fully automated methodology on the number of bugs identified and the bugs to target ratio percentage. On the other hand, the semi-automated method of FTG largely underperforms compared to manual methods and fully automated methods even with the only minor effort required by the software tester.

Measurement of false positives is another metric that we gathered during this experiment. The semi-automated method of FTG showed the most number of false positives (71) followed by manually developed fuzz drivers (50) and fully automatically generated fuzz drivers (12). The high false-positive rate of FTG is largely due to its inability to correctly identify the function signature of the target function. This was the same for the other two fuzz driver development techniques but they are not as much affected as FTG in terms of the results they performed in this experiment.

Our analysis shows that false positives occur due to incorrect memory allocation and incorrect identification of parameters and their sizes. As a result, memory violation errors occurred on the fuzz driver itself, thus, causing crashes.

In the case of manually written fuzz drivers, false positives occur due to the software tester's lack of deep knowledge of certain techniques or simply down to human error. This result validates our findings in Section 2.6.10, where we identified that lack of domain knowledge causes faulty fuzz drivers.

For the semi-automated technique of FTG, false positives occur mainly due to `@fuzztest` directive not being able to correctly inform the fuzz driver generation process about the correct functionality of the target function. The authors Kelly et al. [73] propose clean up parameters (Section 3.2.1) to manage memory, however, if the complexity of the function is not correctly identified or if FTG does not recognise the function signature correctly, the effort provided by cleanup parameters are not as effective, as we identified through a high number of false positives for FTG.

The false positives from FuzzGen fuzz drivers are also due to FuzzGen not identifying function signatures correctly. However, false positives of FuzzGen are five times less than manually developed fuzz drivers and six times less than semi-automatically generated fuzz drivers through FTG. This makes the fully automated fuzz driver generation method the most effective in terms of not producing false positives.

Overall, the bug identification results entail that manually developed fuzz drivers perform best in terms of identifying bugs but the fully automated method of FuzzGen is remarkably close with less false positive generation. The fully automated method of FuzzGen has the most success rate per effort compared to the other two methods. On the other hand, the semi-automated method of FTG largely underperforms in terms of identifying bugs and producing extremely high levels of false positives.

3.4.3 Analysis of code coverage

Similar to the bug identification, we analysed the code coverage of target functions when they are fuzzed using all three fuzz driver development methods (Table 3.4).

TABLE 3.4. Code coverage.

Fuzz Driver Development Method	Average Code Coverage
Manually Developed Fuzz Driver	152.25
FTG	11.31
FuzzGen	131.85

It is evident from our results that on average, manually developed fuzz drivers cover more blocks of code compared to fuzz drivers generated through FTG and FuzzGen. Both manually developed fuzz drivers and fully automatically generated fuzz drivers greatly outperform the semi-automatic method by showing more than ten times greater code coverage. Manually developed fuzz drivers show slightly more code coverage compared to fully automatic fuzz drivers. In the research work of Ispoglou et al. [62], they claim that the fully automatic method of FuzzGen outperforms manually developed fuzz drivers by 6.94%. However, our findings indicate that FuzzGen shows 15.5% less code coverage compared to fully manual techniques. These results implicate that writing a fuzz driver manually with the knowledge regarding the codebase helps to outperform the semi-automated and automated fuzz driver generation techniques in terms of code coverage.

The fully automated techniques of FuzzGen outperform semi-automatic techniques of FTG by 11.66 times. This is mainly due to the adoption of A^2DG graph for path identification and function dependency identification in FuzzGen as explained in Section 3.2.2. FTG only identifies the function signature and it does not identify any information regarding the internal structure of the target function similar to the fuzz drivers generated through FuzzGen or software testers' knowledge in the target code such as in manually developed fuzz drivers. Thus, the low code coverage of FTG. Furthermore, FTG only seems to identify one function to fuzz while FuzzGen and manually developed fuzz drivers have the capacity to identify further functions that are connected to the target functions, thus showcasing another reason why FTG underperforms in terms of code coverage.

When comparing the code coverage results with bug identification, it is clear that manually developed fuzz drivers outperform fully automated fuzz driver development methods and semi-automated fuzz driver development methods. Semi-automated methods of FTG show the worst results for both bug identification and code coverage due to its lack of function signature identification and lack of internal function structure identification (control flow and data flow). When it comes to the code coverage, we can see in Table 3.4 that manual methods outperform fully automated methods by 15.5%, however, when it comes to bug identification, there is only 1.5% increase in performance in manually developed fuzz drivers compared to the fully automated method. This implicates that even with lower code coverage, FuzzGen (fully automated) closely matches manually developed fuzz drivers.

3.4.4 Analysis of code complexity measures

After the evaluation of fuzzer performance, we analysed how the cyclomatic complexity and the Halstead volume of the target program affect the performance of three different types of fuzz drivers in the fuzzing campaign. In this step of the experiment, we made a slight modification to our sample of 148 results. To directly compare these three sets of fuzz drivers, we had to make sure that we select a sample where all three drivers are compatible with the same target programs to nullify any biases or errors due to a different number of compatible projects. Therefore, once we applied this filter, our sample of 148 projects was reduced to 87 projects (Figure 3.6). The results in the following sections are based on these 87 projects.

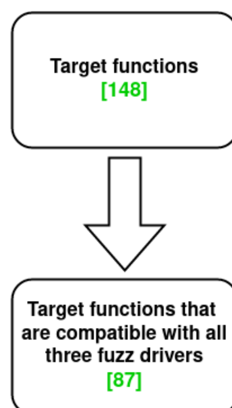

FIGURE 3.6. The number of target functions that are compatible with all three fuzz driver development techniques.

Analysis of cyclomatic complexity

Initially, we analysed how the code coverage is affected when the cyclomatic complexity increases. Therefore, we measured the cyclomatic complexity of all 87 target functions and we compared the correlation between the code coverage and the cyclomatic complexity for fuzzing campaigns of manually developed fuzz drivers, semi-automatically developed fuzz drivers (FTG) and fully automatic fuzz drivers (FuzzGen).

Our results show that the semi-automated method of FTG does not correlate with an increase in cyclomatic complexity (Figure 3.8). This is mainly due to its limitation of only targeting one function and not identifying related functions and function dependencies in a file. However, there is a positive correlation between the code coverage and cyclomatic complexity for both manually developed fuzz drivers (Figure 3.7) and fully automatically generated fuzz drivers (FuzzGen) (Figure 3.9). The slope value of Figure 3.7 for manually developed fuzz drivers is 1.77 compared to the slope value of 1.76 for fully automatically developed fuzz drivers. However, the 0.01 difference in slope values suggests that the fully automated method of FuzzGen has a similar rate of increase in code coverage compared to manual techniques considering the effort factor that takes to write a fuzz driver for highly complex code. Moreover, there is a positive correlation coefficient and coefficient of determination for both manually developed fuzz drivers ($p = 0.77$, $r^2 = 0.59$) and automatically generated fuzz drivers through FuzzGen ($p = 0.92$, $r^2 = 0.84$) when comparing with cyclomatic complexity and the results show that strength of the correlation is much higher for FuzzGen fuzz drivers for this sample data set.

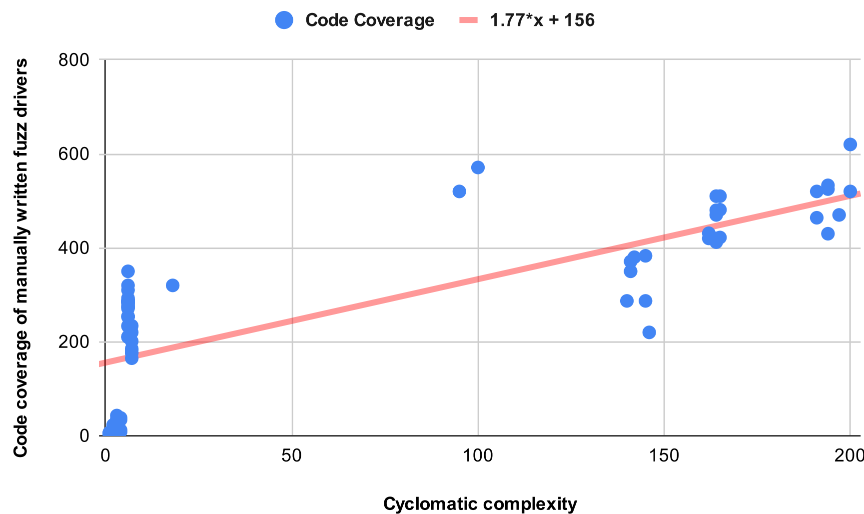

FIGURE 3.7. Cyclomatic complexity vs code coverage of manually written fuzz drivers.

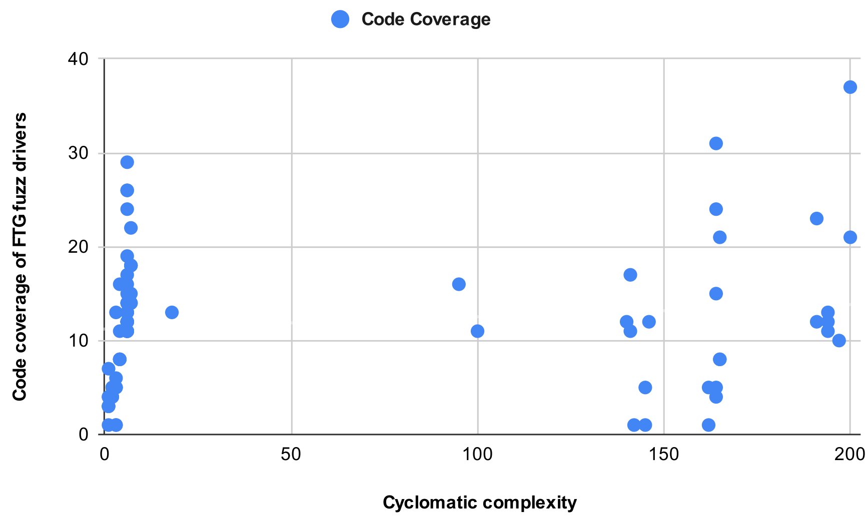

FIGURE 3.8. Cyclomatic complexity vs code coverage of FTG fuzz drivers.

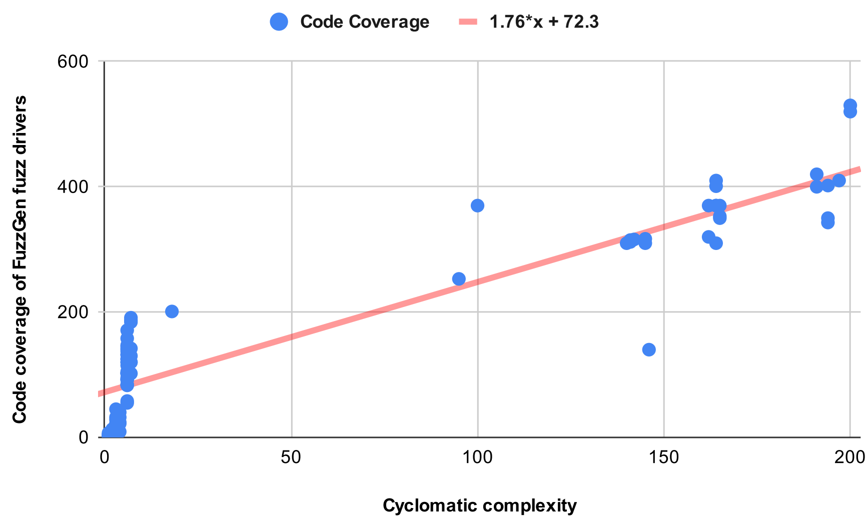

FIGURE 3.9. Cyclomatic complexity vs code coverage of FuzzGen fuzz drivers.

As we identified in Section 3.2.5, multiple research studies [76, 144] show that when the cyclomatic complexity is above 50, the code is too complex and difficult to test. We tried to evaluate this concept with our results by checking how the average code coverage change when the cyclomatic complexity goes above 50 for target functions.

Table 3.5. Percentage increase in average code coverage when cyclomatic complexity (CC) of the target function goes beyond 50.

Fuzz driver type	Code coverage when CC $<$ 50	Code coverage when CC $>$ 50	Percentage increase
Manual	164.53	444.70	170.30%
FTG	11.52	12.85	11.55%
FuzzGen	81.57	357.11	337.62%

The results in Table 3.5 show average code coverage of fuzz drivers when the cyclomatic complexity of the target code is less than 50 and more than 50. Also, we calculated the percentage increase.

In general, the average code coverage due to manually developed fuzz drivers is greater than both automatic fuzz driver development techniques and semi-automatic fuzz driver development techniques. This implies that human input in identifying program paths and dependencies still slightly outperforms the automatic program path identification of FuzzGen through the construction of A^2DG graph to aid code coverage against the cyclomatic complexity.

The percentage increase in code coverage in the fully automated method of FuzzGen shows the largest increase in code coverage by 337.62% when the cyclomatic complexity of the target code goes beyond 50. We identified the reason for this through an experimentation process. The work of Ispoglou et al. [62] proposes FuzzGen for large codebases with multiple functions. Therefore, for large codebases, it generates A^2DG graph and analyses the API structure as explained in Section 3.2.2. However, when the target program is only a single function or very few functions, the process of control flow and dependency analysis is ignored, thus, the low code coverage when the cyclomatic complexity is low. This means that when the target code size is low, the improvements in code coverage are only dependent on the seed modification technique of `EatData` function as explained in Section 3.2.2. Hence, measures should be taken to improve the quality of the fuzz driver developed by FuzzGen for less complex or smaller functions for better code coverage.

Analysis of Halstead metrics

There is no correlation between the code coverage of FTG fuzz drivers and Halstead volume (Figure 3.11). The answer to this phenomenon lies within the way how FTG is programmed to function. When the software tester selects a function with the predefined comment as explained in Section 3.2.1, the tester does not give any information regarding the internal parameters (which would become operands) nor their types to FTG. FTG is programmed only to identify the content in the function declaration and their types; it does not identify other parameters or content within the target function. Since it does not know parameters within the function, it shows no correlation to Halstead volume. As a result, it lacks a thorough knowledge regarding the data flow within the function, hence resulting in low code coverage compared to manually developed fuzz drivers and automatically developed fuzz drivers.

To improve FTG code coverage performance on this occasion, it should be modified to identify parameters within the body of the function when developing fuzz drivers. Parameters in the function can be identified by scanning the whole function rather

than stopping scanning at the function signature to develop the fuzz driver. This process will allow the fuzz driver to get a good understanding of all the operands and their types, thus enhancing its knowledge regarding the operation and data flow within the function. As a result, the code coverage will improve.

There is a positive correlation between the increase in Halstead volume and code coverage for both manually developed fuzz drivers (Figure 3.10) and automatically developed fuzz drivers (Figure 3.12). Our results show that the slope value of FuzzGen in Figure 3.12, which is 0.0027 is very similar to or slightly higher than the slope value of manually developed fuzz drivers in Figure 3.10 (0.0026). This result is quite remarkable since it shows that the fully automated method matches or has a slightly higher rate of data flow understanding regarding the parameters within the target code. Moreover, there is a positive correlation coefficient and coefficient of determination for both manually developed fuzz drivers ($p = 0.74$, $r^2 = 0.55$) and automatically generated fuzz drivers through FuzzGen ($p = 0.90$, $r^2 = 0.82$) when comparing with Halstead Volume and the results show that strength of the correlation is much higher for FuzzGen fuzz drivers for this sample data set.

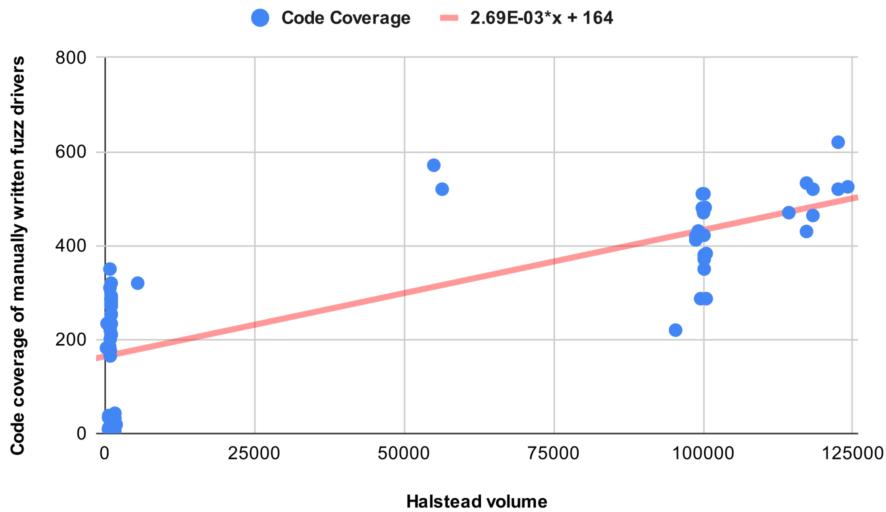

FIGURE 3.10. Halstead volume vs code coverage of manually written fuzz drivers.

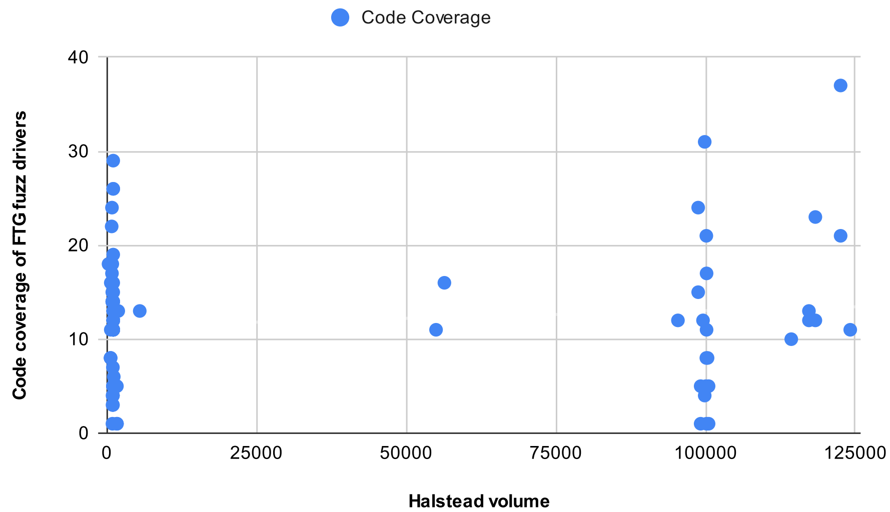

FIGURE 3.11. Halstead volume vs code coverage of FTG fuzz drivers.

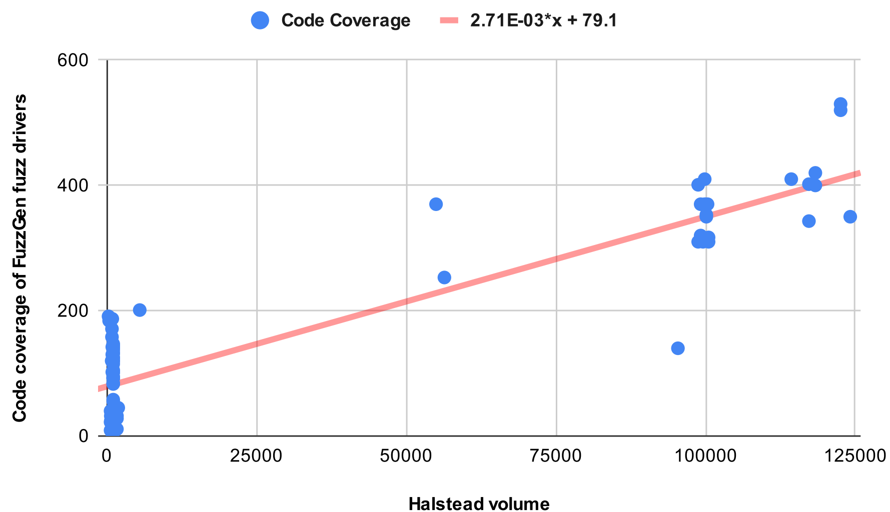

FIGURE 3.12. Halstead volume vs code coverage of FuzzGen fuzz drivers.

This is potentially due to the automatic generation of an information file called "meta file" (Section 3.2.2) by FuzzGen. FuzzGen scans all the files of the target project and creates a file with information including the names of all the functions, parameters, signed parameters, includes etc. Through this file, FuzzGen understands all the information regarding potential parameters and in which functions they appear and the order in which they would appear within a function. The automatic generation of meta files largely reduces the time consuming human effort of trying to recognise parameters and their types in the target code.

The average code coverage of the manually written fuzz drivers is higher than the code coverage from the fully automatic fuzz drivers from FuzzGen as we can see from the Y axis in Figure 3.10 and Figure 3.12. Also, as we identified in Section 3.4.3, manually written fuzz drivers outperform fully automatically developed fuzz drivers by 15.5%. Therefore, it can be argued that on average human input still surpasses

automatic scanning of the code when it comes to understanding the operations and data flow within the function.

However, as we identified in Section 3.4.2, the percentage of the target to bug ratio of fully automated methods from FuzzGen is closely matched with manually written fuzz drivers. This implies the efficiency of FuzzGen in identifying a high amount of bugs for lesser code coverage. Halstead volume results clearly implicate the reasoning behind this phenomenon by proving that FuzzGen has the ability to outperform other techniques through the better identification of the program data flow. Thus, giving it an advantage in recognising bug locations compared to the other two techniques.

From the above results, it is clear that if FuzzGen can cover more code with further improvements; it can outperform manually written fuzz drivers. In Section 3.4.2 and Section 3.4.1, we recognised that a reason behind the reduced code coverage of FuzzGen is due to its lack of understanding to convert all types of function signatures into valid fuzz drivers. Therefore, improvements in function signature identification in FuzzGen will guarantee better code coverage and an increased amount of bug identification, thus, surpassing the performance of both manually written fuzz drivers and semi-automatically generated fuzz drivers.

3.5 Limitations and threats to validity

We used GitHub to collect open-source projects that already had manually written fuzz drivers. These fuzz drivers are not written by a single software engineer, hence when using these fuzz drivers we deal with source code that is written by testers with multiple levels of domain knowledge. Hence, we collected fuzz drivers that were well written and also poorly written by software testers with varying abilities. All the other studies that have done comparisons [7, 62, 73, 183] had a single software tester writing the fuzz driver. However, we believe that this would introduce human bias to results. On the other hand, our sample depicts a clear representation of how software testers with varying abilities would write fuzz drivers for target functions by nullifying human bias.

There were a high amount of false positives and compilation errors when running some of the open-source projects on GitHub. This may be due to the lack of quality control mechanisms of GitHub when accepting projects. We only ran software products that compiled correctly with fuzz drivers.

Two other research studies that boast to have similar results to FuzzGen or better results to FuzzGen are Fudge [7] and Intelligen [183]. We discussed their functionality in Section 3.2.3, however, we could not test these two tools since they are not available as open-source projects for results replication. As a result, we only used FuzzGen to generate fuzz drivers automatically. Of all the state of the art projects that propose automatic fuzz driver generation, FuzzGen is the only open-source project that is available for research purposes. Therefore, to measure the effectiveness of automatic fuzz driver generation against manually developed fuzz drivers, we are restricted to one open-source project.

3.6 Summary

In this chapter, we report on the case study and the empirical analysis that we carried out to identify the effectiveness of three different fuzz driver development strategies: manually developed fuzz drivers, semi-automatically generated fuzz drivers with human-in-the-loop approaches and fully automatically generated fuzz drivers. We

used 148 open-source software projects for this experimental analysis and we summarise,

- Manually developed fuzz drivers outperform semi-automatically developed fuzz drivers with human-in-the-loop approaches and automatically developed fuzz drivers in terms of bug identification. Manually developed fuzz drivers find 8 more bugs compared to the fully automated method from FuzzGen. It is important to note that the semi-automated fuzz driver development technique largely underperforms against the other two techniques by identifying more than four times fewer bugs.

- Fully automated fuzz driver generation of FuzzGen gives a higher success rate per effort in identifying bugs compared to manually developed fuzz drivers and semi-automatic fuzz driver generation of FTG.

- Fully automated fuzz driver generation of FuzzGen identifies the least number of false positives (12) followed by manually developed fuzz drivers (50) and the semi-automated method of FTG (71).

- Manually developed fuzz drivers have the highest code coverage followed by fully automated fuzz driver development methods and semi-automated fuzz driver development methods

- Manually developed fuzz drivers show higher code coverage as the cyclomatic complexity increases followed by fully automated fuzz driver generation. Semi-automated fuzz driver generation of FTG shows no effect on an increase in cyclomatic complexity.

- When the cyclomatic complexity of the target code goes beyond its acceptable threshold (beyond 50), fully automated fuzz driver generation of FuzzGen performs better in code coverage compared to manually developed fuzz drivers due to the adoption of control flow analysis and data flow analysis of FuzzGen.

- The code coverage of the semi-automated method of FTG shows no correlation to the increase in Halstead volume due to its lack of ability to identify the data flow within target functions.

- Fully automated fuzz driver generation has a slightly higher rate of increase in code coverage when the Halstead volume increases compared to manually developed fuzz drivers. This is due to the generation of the "meta file" (Section 3.2.2) by FuzzGen with information regarding code parameters and function dependencies.

- Fully automated fuzz driver generation of FuzzGen shows a higher level of bug identification for lower code coverage. However, its overall performance is still less than manually developed fuzz drivers in terms of the total number of bugs identified and code coverage.

Overall, in this chapter, we designed a replicable systematic case study to identify the best state of the art fuzz driver generation method. We tested three state of the art fuzz driver development methods and carried out an empirical analysis to monitor their performance. We recognised, evaluated and reported the reasoning behind performance fluctuations in bug identification and code coverage, thus answering the second research question.

Furthermore, we empirically analysed the effect of the increase in code complexity of the target code and fuzz driver performance in the fuzzing campaign. We monitored how three different types of fuzz drivers perform under different code complexity levels. Then we recognised the reasoning behind their performance by statically and experimentally analysing the development strategies of fuzz driver development methods. Finally, we recognised multiple strategies to improve the state of the art fuzz driver development techniques for better results. Thus, we answered the third research question of this thesis.

Chapter 4

Conclusion and future work

4.1 Conclusion

Fuzz driver development is an integral part of the fuzzing campaign because it acts as the binder between the fuzzer and the target program. When the fuzzer sends random input, the fuzz driver should have the ability to efficiently direct that input to relevant sections of the target code to identify software vulnerabilities. In this thesis, our motivation was to analyse, the role of fuzz driver development in fuzzing campaigns, state of the art fuzz driver generation strategies and the changes in the target code complexity on the fuzz driver performance.

A systematic literature review (SLR) was carried out to analyse and evaluate the state of the art in fuzz driver development. Through this study, we answered the first research question by observing the main challenges, strategies and common practices of fuzz driver development. We critically analysed how fuzz drivers appeared in research studies since 1990 and how they gradually evolved over time with the invention of new techniques and methodologies. This study explored the importance of fuzz drivers in the domain of fuzzing, fuzz driver life cycle, fuzz driver generation techniques, fuzz driver evaluation methodologies and tools to aid the fuzz driver development process.

A case study was carried out to compare the performance of three fuzz driver development techniques: manually written fuzz drivers, semi-automatically developed fuzz drivers with human-in-the-loop approaches and automatically developed fuzz drivers. We implemented a replicable systematic process to collect relevant manually written fuzz drivers from open-source repositories on GitHub. Through data gathering, 148 manually generated fuzz drivers were collected from open-source projects. Then, we generated subsequent semi-automatic and automatic fuzz drivers using FTG [73] and FuzzGen [62]. The case study analysed the performance of multiple fuzz driver techniques in terms of bug identification and code coverage. This study allowed the identification of the best state of the art fuzz driver generation strategy for a successful fuzzing campaign that is available for software testers and researchers as open-source products. Through this study, we successfully answered our second research question.

An empirical study was carried out to identify the change in behaviour of fuzzing campaigns through different fuzz drivers when the code complexities increase in the target program. We explored the effect of the increase in cyclomatic complexity [100] and Halstead volume [54] of the target code on the code coverage of the fuzzing campaign. A thorough analysis was carried out to explore how different types of fuzz drivers would fare with the increase in code complexities. Hence, the reasoning behind the performance levels of fuzz drivers was identified when fuzzing complex code. Furthermore, areas of improvement for the state of the art fuzz driver generators were identified to enhance their performance in fuzzing campaigns for complex code. Through this study, we successfully answered our third research question.

4.2 Future directions

We believe that this thesis contributes to identifying the impact of fuzz drivers in fuzzing research. We also identified multiple areas in the field of fuzz drivers that can be explored in the future, which are listed below.

4.2.1 Novel ways to improve information transfer from function signatures to a fuzz driver

Our results in Chapter 3 prove that most tools can identify all types of function signatures but they cannot convert all the identified function signatures into valid fuzz drivers [62, 73, 183]. For example, when there are multiple pointers in the function signature, both FTG and FuzzGen fails to generate valid fuzz drivers. Hence, there is a gap for further research in this area.

FTG is only compatible with simple function signatures and it has good coverage within those functions. Therefore, the ability to identify complex function signatures and the ability to convert those into fuzz drivers will further enhance the code coverage and bug identification of FTG. The inclusion of AST monitoring of the source code would improve function identification for FTG.

The requirement is slightly different for automatically developed fuzz drivers of FuzzGen. Our results from Chapter 3 clearly show that if FuzzGen could convert all the function signatures that it identifies into valid fuzz drivers, it would have had the ability to outperform manually developed fuzz drivers in bug identification and code coverage due to its less false positives ratio per target. Therefore, measures should be taken to improve the conversion of identified function signature to a valid fuzz driver in FuzzGen.

There is room for improvement by observing areas such as documentation files and code comments when extracting information about a specific method or a function. For example, the use of standardised documentation generators such as JavaDoc [66] for Java programs and DoxyGen [35] for C, and C++ programs and then using the documentation information to gather further information regarding functions when generating the fuzz driver. If more intricate details of the function are available, then there is a high chance of correctness in the fuzz driver output.

4.2.2 Novel ways to automate unit test to fuzz driver conversion

The automation of the conversion of unit tests to fuzz drivers is an area that needs more novelty. This is still a manual task in many research studies except for the work proposed by Jang and Kim [65] and Myllylahti [109]. The methodologies proposed by these two studies scan the unit test, identify elements of the target code through the content of the unit test and generate the fuzz driver. This strategy could be further improved by gathering the AST of the target code and using program slicing to recognise the exact features of the code that requires fuzzing similar to the methodology followed by Babić et al. [7]. Thus, the use of gathered information about the target code along with features identified from unit tests will generate a valid fuzz driver.

4.2.3 Minimising human effort in semi-automated fuzz driver development

There is still a requirement to have domain knowledge about the codebase when the fuzz driver generation process is semi-automatic. In Kelly et al. [73] work of FTG, which comments on top of the target function, the tester has to manually give

information regarding the function declaration types and features (e.g. when it is an array) in the comment as the complexity of the target function increases. There is also the requirement to provide garbage collection information to FTG. The commenting of `@fuzztest` would only work on simple function signatures and when the function signatures get complex, the commenting requires a certain level of knowledge regarding the codebase. Therefore, measures should be taken to improve this scenario, hence only the commenting of `@fuzztest` is enough to generate a fuzz driver for any type of function signature.

4.2.4 Inclusion of single function fuzz driver generation capacity for state of the art automatic fuzz driver generators

Automatic fuzz driver generators such as FuzzGen [62], Fudge [7] and Intelligen [183] focus on generating fuzz drivers to all the functions in the target codebase. However, if the developer wants to fuzz a single function or a handful of functions, the service is not available from these three fuzz driver generators. This service is available through FTG [73] but it is not as efficient in identifying bugs or code coverage as it was identified in Chapter 3. Therefore, modifications should be made for automatic fuzz driver generators such as FuzzGen, Intelligen and Fudge to allow the developer to point at a single function, multiple functions or a single source code file to generate fuzz drivers. Normally, the motivation of fuzzing is to test as many code of the target program as much as possible, however, with the introduction of FTG [73] and proposed software projects such as CodeIntelligence [28] and Jazzer [67], single program fuzzing is taking a new leap in research focus. Therefore, this direction of adoption could enhance the appeal of the state of the art fuzz driver generators.

4.2.5 Development of fuzz drivers for multiple libraries

Automatic fuzz drivers such as FuzzGen [62] focus on fuzzing single libraries and do not account for the interaction between multiple libraries. There is room to extend FuzzGen to build fuzz drivers for multiple libraries that are connected. Therefore, complex inter-dependencies between libraries should be explored through the automated fuzz driver development tool.

4.2.6 Comparison of micro fuzzing and fuzzing campaigns with fuzz drivers

Micro Executions is a testing strategy proposed by Godefroid [47] as mentioned in section Section 2.7.2. Micro execution allows the execution of the code without a test driver or input data. The software tester should identify the location to be tested in dll or exe. Then the code executes for testing purposes at that location using a run time virtual machine. This technique further appears in fuzzing research in the tool HotFuzz [11] as micro-fuzzing. As a result, micro fuzzing carries out fuzzing campaigns without using fuzz drivers. There is no comparison between the effectiveness of this strategy and a fuzzing campaign through fuzz drivers according to our findings in Chapter 2, thus it is a new area of research to be explored.

4.2.7 Fuzz driver development automation for multiple programming languages

All of the research studies that automated fuzz driver development focused on a single programming language. Therefore, there is potential to develop a research study

with a tool that would automatically generate fuzz drivers for multiple languages and multiple fuzzers. CI Fuzz claims to have this ability in Section 2.9 but there is no supporting evidence or results for this process other than a demo video. Furthermore, this tool is not part of our attribute matrix as explained in Section 2.9. Therefore, more research should be carried out in such research areas to improve existing tools available for fuzz driver development automation. However, it could be a challenge to develop a single product that is compatible with multiple programming languages due to the different structures and features that these different languages possess. Therefore, to successfully achieve this feature, a language-independent specification approach [12, 20, 87] could be adopted by researchers.

4.2.8 More applications of function importance ranking algorithms

Intelligen [183] proposes an algorithm to identify function importance and discard less important functions when developing fuzz driver as we identified in Section 2.7.2. Intelligen only focuses on C and C++. Therefore, a similar approach could be applied when generating fuzz drivers for other programming languages to recognise relevant function signatures and to promote efficient code coverage.

Chapter 5

Appendix A: Selected studies in the systematic literature review

Table A. Selected studies in the systematic literature review.

Title	Authors	Year	Reference
Flayer: Exposing application internals	Drewry, Will; Ormandy, Tavis	2007	[36]
Grammar-based whitebox fuzzing	Godefroid, Patrice; Kiezun, Adam; Levin, Michael Y.	2008	[48]
Configuring resource managers using model fuzzing: A case study of the. NET thread pool	Hellerstein, Joseph L.	2009	[57]
A symbolic execution framework for javascript	Saxena, Prateek; Akhawe, Devdatta; Hanna, Steve; Mao, Feng; McCamant, Stephen; Song, Dawn	2010	[134]
D4. 3 Final report on inspection methods and prototype vulnerability recognition tools	Fraunh, Get; Liu, M. I.; Per H; Meland, SINTEF; Raiteri, Fabio	2010	[41]
Detecting atomic-set serializability violations in multithreaded programs through active randomized testing	Fraunh, Get; Liu, M. I.; Per H; Lai, Zhifeng; Cheung, Shing-Chi; Chan, Wing Kwong	2010	[83]
Effective detection of atomic-set serializability violations in multithreaded programs	Lai, Zhifeng	2010	[82]
Detect Program Vulnerabilities Using Trace-based Security Testing	Zhang, Dazhi	2011	[182]
Effective code coverage in compositional systematic dynamic testing	Wan, Zhiyuan; Zhou, Bo	2011	[169]
Comparative Language Fuzz Testing	Louridas, Diomidis Spinellis Vassilios Karakoidas Panos	2012	[96]
Implementation and testing of a blackbox and a whitebox fuzzer for file compression routines	Tobkin, Toby	2013	[160]
Incorporating fuzz testing to unit testing regime	Myllylahti, Juho	2013	[109]
A Smart Fuzzing Approach for Integer Overflow Detection	Cai, Jun; Zou, Peng; He, Jun; Ma, Jinxin	2014	[19]
Micro execution	Godefroid, Patrice	2014	[47]
Analyzing Distributed Multi-platform Java and Android Applications with ShadowVM	Sun, Haiyang; Zheng, Yudi; Bulej, Lubomír; Kell, Stephen; Binder, Walter	2015	[154]
Combining static and dynamic analyses for vulnerability detection: Illustration on heartbleed	Kiss, Balázs; Kosmatov, Nikolai; Pariente, Dillon; Puccetti, Armand	2015	[78]
Generating Succinct Test Cases Using Don't Care Analysis	Nguyen, Cuong; Yoshida, Hiroaki; Prasad, Mukul; Ghosh, Indradeep; Sen, Koushik	2015	[111]

Continued on next page

Appendix A – continued from previous page

Title	Authors	Venue	Year
Proving memory safety of the ANI Windows image parser using compositional exhaustive testing	Christakis, Maria; Godefroid, Patrice	2015	[25]
Reins to the Cloud: Compromising Cloud Systems via the Data Plane	Thimmaraju, Kashyap; Shastry, Bhargava; Fiebig, Tobias; Hetzelt, Felicitas; Seifert, Jean-Pierre; Feldmann, Anja; Schmid, Stefan	2016	[159]
Achieving High Coverage for Floating-point Code via Unconstrained Programming (Extended Version)	Fu, Zhoulai; Su, Zhendong	2017	[43]
Exploitability assessment with TEASER	Ulrich, Frederick	2017	[163]
Static Exploration of Taint-Style Vulnerabilities Found by Fuzzing	Shastry, Bhargava; Maggi, Federico; Yamaguchi, Fabian; Rieck, Konrad; Seifert, Jean-Pierre	2017	[143]
WiFuzz: detecting and exploiting logical flaws in the Wi-Fi cryptographic handshake	Vanhoef, Mathy	2017	[166]
Automatically testing implementations of numerical abstract domains	Bugariu, Alexandra; Wüstholz, Valentin; Christakis, Maria; Müller, Peter	2018	[15]
Compiler assisted vulnerability assessment	Shastry, Bhargava	2018	[142]
Computing homomorphic program invariants	Holland, Benjamin Robert	2018	[58]
DeepState: Symbolic unit testing for C and C++	Goodman, Peter; Groce, Alex	2018	[50]
Differential program analysis with fuzzing and symbolic execution	Noller, Yannic	2018	[114]
Fuzz testing in practice: Obstacles and solutions	Liang, Jie; Wang, Mingzhe; Chen, Yuanliang; Jiang, Yu; Zhang, Renwei	2018	[90]
rev. ng: A multi-architecture framework for reverse engineering and vulnerability discovery	Di Federico, Alessandro; Fezzardi, Pietro; Agosta, Giovanni	2018	[31]
RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs	Laeufer, Kevin; Koenig, Jack; Kim, Donggyu; Bachrach, Jonathan; Sen, Koushik	2018	[81]
STADS: Software testing as species discovery	Böhme, Marcel	2018	[16]
A Case Study on Automated Fuzz Target Generation for Large Codebases	Kelly, Matthew; Treude, Christoph; Murray, Alex	2019	[73]
An Efficient Greybox Fuzzing Scheme for Linux-based IoT Programs Through Binary Static Analysis	Zheng, Yaowen; Song, Zhanwei; Sun, Yuyan; Cheng, Kai; Zhu, Hongsong; Sun, Limin	2019	[187]
Analysing the Signal Protocol	van Dam, Dion	2019	[164]
Compositional fuzzing aided by targeted symbolic execution	Ognawala, Saahil; Kilger, Fabian; Pretschner, Alexander	2019	[118]
Deepfuzzer: Accelerated deep greybox fuzzing	Liang, Jie; Jiang, Yu; Wang, Mingzhe; Jiao, Xun; Chen, Yuanliang; Song, Houbing; Choo, Kim-Kwang Raymond	2019	[91]
Defending In-process Memory Abuse with Mitigation and Testing	Chen, Yaohui	2019	[23]
Deterring attackers by Injecting Unexploitable Bugs	Vijtiuk, Juraj	2019	[167]

Continued on next page

Appendix A – continued from previous page

Title	Authors	Venue	Year
DiffFuzz: differential fuzzing for side-channel analysis	Nilizadeh, Shirin; Noller, Yannic; Pasareanu, Corina S.	2019	[113]
Efficient approach to fuzzing interpreters	Dominiak, Marcin; Rauner, Wojciech.	2019	[34]
FirmFuzz: automated IoT firmware introspection and analysis	Srivastava, Prashast; Peng, Hui; Li, Jiahao; Okhravi, Hamed; Shrobe, Howard; Payer, Mathias	2019	[149]
Formal specification and testing of QUIC	McMillan, Kenneth L.; Zuck, Lenore D.	2019	[101]
From proof-of-concept to exploitable	Wang, Yan; Wu, Wei; Zhang, Chao; Xing, Xinyu; Gong, Xiaorui; Zou, Wei	2019	[172]
Fudge: fuzz driver generation at scale	Babić, Domagoj; Bucur, Stefan; Chen, Yaohui; Ivančić, Franjo; King, Tim; Kusano, Markus; Lemieux, Caroline; Szekeres, László; Wang, Wei	2019	[7]
FuzzBuilder: automated building greybox fuzzing environment for C/C++ library	Jang, Joonun; Kim, Huy Kang	2019	[65]
Fuzzing Janus for Fun and Profit	Amirante, Alessandro; Castaldi, Tobia; Miniero, Lorenzo; Romano, Simon Pietro; Saviano, Paolo; Toppi, A.	2019	[5]
Fuzzing Universal Plug and Play	Van Looy, Eleanor	2019	[165]
Fuzzing Universal Plug and Play (UPnP)	Mestdagh, Steven	2019	[103]
Identifying Software and Protocol Vulnerabilities in WPA2 Implementations through Fuzzing	Lioy, Antonio; Müehlberg, Jan Tobias; Vanhoef, Mathy; Marallo, Graziano	2019	[93]
JQF: coverage-guided property-based testing in Java	Padhye, Rohan; Lemieux, Caroline; Sen, Koushik	2019	[122]
KLUZZER: Whitebox Fuzzing on Top of LLVM	Le, Hoang M.	2019	[84]
Smart greybox fuzzing	Pham, Van-Thuan; Böhme, Marcel; Santosa, Andrew Edward; Caciulescu, Alexandru Razvan; Roychoudhury, Abhik	2019	[128]
TestCov: Robust test-suite execution and coverage measurement	Beyer, Dirk; Lemberger, Thomas	2019	[10]
Toward the analysis of embedded firmware through automated re-hosting	Gustafson, Eric; Muench, Marius; Spensky, Chad; Redini, Nilo; Machiry, Aravind; Fratantonio, Yanick; Balzarotti, Davide; Francillon, Aurélien; Choe, Yung Ryn; Kruegel, Christophe	2019	[53]
VisFuzz: understanding and intervening fuzzing with interactive visualization	Zhou, Chijin; Wang, Mingzhe; Liang, Jie; Liu, Zhe; Sun, Chengnian; Jiang, Yu	2019	[188]
The Art, Science, and Engineering of Fuzzing: A Survey	Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, Maverick Woo	2019	[99]

Continued on next page

Appendix A – continued from previous page

Title	Authors	Venue	Year
AFL++: Combining incremental steps of fuzzing research	Fioraldi, Andrea; Maier, Dominik; Eißfeldt, Heiko; Heuse, Marc	2020	[40]
AFLNet: a greybox fuzzer for network protocols	Pham, Van-Thuan; Böhme, Marcel; Roychoudhury, Abhik	2020	[129]
Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints	Song, Dokyung; Hetzelt, Felicitas; Kim, Jonghwan; Kang, Brent Byunghoon; Seifert, Jean-Pierre; Franz, Michael	2020	[147]
Algorithmic improvements for feedback-driven fuzzing	Aschermann, Cornelius	2020	[6]
BaseSAFE: baseband sanitized fuzzing through emulation	Maier, Dominik; Seidel, Lukas; Park, Shinjo	2020	[98]
Better Robustness Testing for Autonomy Systems	Zizyte, Milda	2020	[189]
Building Secure and Reliable Systems	Beyer, Betsy	2020	[9]
Coverage-guided binary fuzzing with rev. ng and llvm libfuzzer	Frighetto, Antonio; Agosta, Giovanni; Di Federico, Ph D. Alessandro; Gussoni, M. Sc Andrea	2020	[42]
Deep learning for compilers	Cummins, Christopher Edward	2020	[30]
Edge of the Art in Vulnerability Research	Tenaglia, Scott; Adams, Perri	2020	[158]
Efficient Dependability Assessment of Systems Software	Coppik, Nicolas	2020	[29]
Fuzz4B: a front-end to AFL not only for fuzzing experts	Miyaki, Ryu; Yoshida, Norihiro; Tsuzuki, Natsuki; Yamamoto, Ryota; Takada, Hiroaki	2020	[107]
Fuzzgen: Automatic fuzzer generation	Ispoglou, Kyriakos; Austin, Daniel; Mohan, Vishwath; Payer, Mathias	2020	[62]
Fuzzing binaries for memory safety errors with QASan	Fioraldi, Andrea; D'Elia, Daniele Cono; Querzoni, Leonardo	2020	[40]
Fuzzing JavaScript Engines with Aspect-preserving Mutation	Park, Soyeon; Xu, Wen; Yun, Insu; Jang, Daehee; Kim, Taesoo	2020	[125]
Fuzzing: On the exponential cost of vulnerability discovery	Böhme, Marcel; Falk, Brandon	2020	[17]
Greybox Automatic Exploit Generation for Heap Overflows in Language Interpreters	Heelan, Sean	2020	[56]
HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing	Blair, William; Mambretti, Andrea; Arshad, Sajjad; Weissbacher, Michael; Robertson, William; Kirda, Engin; Egele, Manuel	2020	[11]
HyDiff: Hybrid differential software analysis	Noller, Yannic; Păsăreanu, Corina S.; Böhme, Marcel; Sun, Youcheng; Nguyen, Hoang Lam; Grunske, Lars	2020	[115]
IFFSET: in-field fuzzing of industrial control systems using system emulation	Tychalas, Dimitrios; Maniatakos, Michail	2020	[162]
Magma: A Ground-Truth Fuzzing Benchmark	Hazimeh, Ahmad; Herrera, Adrian; Payer, Mathias	2020	[55]

Continued on next page

Appendix A – continued from previous page

Title	Authors	Venue	Year
MoFuzz: A Fuzzer Suite for Testing Model-Driven Software Engineering Tools	Nguyen, Hoang Lam; Nassar, Nebras; Kehrer, Timo; Grunske, Lars	2020	[112]
Program State Abstraction for Feedback-Driven Fuzz Testing using Likely Invariants	Fioraldi, Andrea	2020	[38]
Property-oriented Model-Based Testing With Fuzzing–Technical Report	Huang, Wen-ling; Krafczyk, Niklas; Le, Hoang M.; Peleska, Jan	2020	[60]
SpecFuzz: Bringing Spectre-type vulnerabilities to the surface	Oleksenko, Oleksii; Trach, Bohdan; Silberstein, Mark; Fetzer, Christof	2020	[119]
SunDew: Systematic Automated Security Testing	Ivančić, Franjo	2020	[63]
The Industrial Age of Hacking	Nosco, Timothy; Ziegler, Jared; Clark, Zechariah; Marrero, Davy; Finkler, Todd; Barbarello, Andrew; Petullo, W. Michael	2020	[116]
Towards making formal methods normal: meeting developers where they are	Reid, Alastair; Church, Luke; Flur, Shaked; de Haas, Sarah; Johnson, Maritza; Laurie, Ben	2020	[132]
Web Protocol Fuzzing of the TLS/SSL Protocol with Focus on the OpenSSL Library	Khalaf, Fouad Nouri	2020	[75]
An Empirical Study of OSS-Fuzz Bugs	Ding, Zhen Yu; Goues, Claire Le	2021	[33]
Efficient Fuzz Testing for Apache Spark Using Framework Abstraction	Zhang, Qian; Wang, Jiyuan; Gulzar, Muhammad Ali; Padhye, Rohan; Kim, Miryung	2021	[185]
Exposing bugs in JavaScript engines through test transplantation and differential testing	Lima, Igor; Silva, Jefferson; Miranda, Breno; Pinto, Gustavo; d'Amorim, Marcelo	2021	[92]
Fuzzing: Challenges and Reflections	Boehme, Marcel; Cadar, Cristian; Roychoudhury, Abhik	2021	[13]
ICSFuzz: Manipulating I/Os and Repurposing Binary Code to Enable Instrumented Fuzzing in ICS Control Applications	Tychalas, Dimitrios; Benkraouda, Hadjer; Maniatakos, Michail	2021	[162]
Industrial Oriented Evaluation of Fuzzing Techniques	Wang, Mingzhe; Liang, Jie; Zhou, Chijin; Chen, Yuanliang; Wu, Zhiyong; Jiang, Yu	2021	[170]
Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing	Wang, Mingzhe; Wu, Zhiyong; Xu, Xinyi; Liang, Jie; Zhou, Chijin; Zhang, Huafeng; Jiang, Yu	2021	[171]
IntelliGen: Automatic Driver Synthesis for Fuzz Testing	Zhang, Mingrui; Liu, Jianzhong; Ma, Fuchen; Zhang, Huafeng; Jiang, Yu	2021	[183]
JMPscare: Introspection for Binary-Only Fuzzing	Maier, Dominik; Seidel, Lukas	2021	[97]
Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types	Schumilo, Sergej; Aschermann, Cornelius; Abbasi, Ali; Wörner, Simon; Holz, Thorsten	2021	[137]
Performant Software Hardening under Hardware Support	Ding, Ren	2021	[32]

Continued on next page

Appendix A – continued from previous page

Title	Authors	Venue	Year
Practical Systems For Strengthening And Weakening Binary Analysis Frameworks	Jung, Jinho	2021	[70]
RULF: Rust Library Fuzzing via API Dependency Graph Traversal	Jiang, Jianfeng; Xu, Hui; Zhou, Yangfan	2021	[69]
SyRust: Automatic Testing of Rust Libraries with Semantic-Aware Program Synthesis–Technical Report	Takashima, Yoshiki; Goncalves Martins, Ruben Carlos; Jia, Limin; Pasareanu, Corina	2021	[155]
Test Automation	Peleska, Jan; Huang, Wenling	2021	[126]
WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning	Jung, Jinho; Tong, Stephen; Hu, Hong; Lim, Jungwon; Jin, Yonghwi; Kim, Taesoo	2021	[71]

Bibliography

[1] Adobe. Binspector: Evolving a security tool. https://blogs.adobe.com/security/2015/05/binspector-evolving-a-security-tool.html/, 2021. Accessed: 2021-11-06.

[2] Md Abdullah Al Mamun, Christian Berger, and Jörgen Hansson. Effects of measurements on correlations of software code metrics. *Empirical Software Engineering*, 24(4):2764–2818, 2019.

[3] Mahmoud Alfadel, Armin Kobilica, and Jameleddine Hassine. Evaluation of halstead and cyclomatic complexity metrics in measuring defect density. In *2017 9th IEEE-GCC Conference and Exhibition (GCCCE)*, pages 1–9. IEEE, 2017.

[4] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[5] Alessandro Amirante, Tobia Castaldi, Lorenzo Miniero, Simon Pietro Romano, Paolo Saviano, and A. Toppi. Fuzzing Janus for Fun and Profit. In *2019 Principles, Systems and Applications of IP Telecommunications (IPTComm)*, pages 1–7. IEEE, 2019.

[6] Cornelius Aschermann. *Algorithmic improvements for feedback-driven fuzzing.* PhD Thesis, Ruhr-Universität Bochum, 2020.

[7] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: fuzz driver generation at scale. In *2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 975–985, 2019.

[8] Mourad Badri and Fadel Toure. Empirical analysis of object-oriented design metrics for predicting unit testing effort of classes. 2012.

[9] Betsy Beyer. *Building Secure and Reliable Systems.* O'Reilly Media, 2020.

[10] Dirk Beyer and Thomas Lemberger. TestCov: Robust test-suite execution and coverage measurement. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1074–1077. IEEE, 2019.

[11] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. *arXiv preprint arXiv:2002.03416*, 2020.

[12] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. Pisa—a platform and programming language independent interface for search algorithms. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 494–508. Springer, 2003.

[13] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and Reflections. *IEEE Softw.*, 38(3):79–86, 2021.

[14] Jonas Bogenberger. Automated fuzz target generation for c libraries. https://www.sec.in.tum.de/i20/student-work/automated-fuzz-target-generation-for-c-libraries, 2021. Accessed: 2021-08-06.

[15] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. Automatically testing implementations of numerical abstract domains. In *33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 768–778, 2018.

[16] Marcel Böhme. STADS: Software testing as species discovery. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(2):1–52, 2018. Publisher: ACM.

[17] Marcel Böhme and Brandon Falk. Fuzzing: On the exponential cost of vulnerability discovery. In *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 713–724, 2020.

[18] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[19] Jun Cai, Peng Zou, Jun He, and Jinxin Ma. A Smart Fuzzing Approach for Integer Overflow Detection. *Information Technology in Industry*, 2(3):98–103, 2014.

[20] Laura M Castro and Miguel A Francisco. A language-independent approach to black-box testing using erlang as test specification language. *Journal of Systems and Software*, 86(12): 3109–3122, 2013.

[21] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. {MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multi-threaded programs. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2325–2342, 2020.

[22] M-H Chen, Michael R Lyu, and W Eric Wong. Effect of code coverage on software reliability measurement. *IEEE Transactions on reliability*, 50(2):165–170, 2001.

[23] Yaohui Chen. *Defending In-process Memory Abuse with Mitigation and Testing*. Northeastern University, 2019.

[24] Bharti Chimdyalwar and Shrawan Kumar. Effective false positive filtering for evolving software. In *Proceedings of the 4th India Software Engineering Conference*, pages 103–106, 2011.

[25] Maria Christakis and Patrice Godefroid. Proving memory safety of the ANI Windows image parser using compositional exhaustive testing. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 373–392. Springer, 2015.

[26] Cisco. Cisco secure development lifecycle. https://www.cisco.com/c/en/us/about/security-center/security-programs/secure-development-lifecycle/sdl-process/validate.html, 2021. Accessed: 2021-08-06.

[27] Code-Intelligence. Ci fuzz software. https://page.code-intelligence.com/sign-up-demo?hsLang=en, 2021. Accessed: 2021-08-06.

[28] Code-Intelligence. Code intelligence software tools. https://www.code-intelligence.com/, 2021. Accessed: 2021-08-06.

[29] Nicolas Coppik. Efficient Dependability Assessment of Systems Software. 2020. Publisher: Technische Universität Darmstadt.

[30] Christopher Edward Cummins. *Deep learning for compilers*. PhD thesis, The University of Edinburgh, 2020.

[31] Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. rev. ng: A multi-architecture framework for reverse engineering and vulnerability discovery. In *2018 International Carnahan Conference on Security Technology (ICCST)*, pages 1–5. IEEE, 2018.

[32] Ren Ding. *Performant Software Hardening under Hardware Support*. PhD Thesis, Georgia Institute of Technology, 2021.

[33] Zhen Yu Ding and Claire Le Goues. An Empirical Study of OSS-Fuzz Bugs. *arXiv preprint arXiv:2103.11518*, 2021.

[34] Marcin Dominiak and Wojciech Rauner. Efficient approach to fuzzing interpreters. *BlackHat Asia*, 2019.

[35] DoxyGen. Doxygen software. https://www.doxygen.nl/index.html, 2021. Accessed: 2021-08-06.

[36] Will Drewry and Tavis Ormandy. Flayer: Exposing application internals. *USENIX Security Symposium*, 2007.

[37] KO Emergy and Brenda K Mitchell. Multi-level software testing based on cyclomatic complexity. In *IEEE National Aerospace and Electronics Conference*, pages 500–507. IEEE, 1989.

[38] Andrea Fioraldi. Program State Abstraction for Feedback-Driven Fuzz Testing using Likely Invariants. *arXiv preprint arXiv:2012.11182*, 2020.

[39] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing binaries for memory safety errors with QASan. In *2020 IEEE Secure Development (SecDev)*, pages 23–30. IEEE, 2020.

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

[41] GET Fraunh, M. I. Liu, Sintef Per Haakon Meland, and Fabio Raiteri. D4. 3 Final report on inspection methods and prototype vulnerability recognition tools. 2010. Publisher: Citeseer.

[42] Antonio Frighetto, Giovanni Agosta, Ph D. Alessandro Di Federico, and M. Sc Andrea Gussoni. Coverage-guided binary fuzzing with rev.ng and llvm libfuzzer. Master's thesis, Master's thesis, Dipartimento di Elettronica, Politecnico Milano, 2020.

[43] Zhoulai Fu and Zhendong Su. Achieving High Coverage for Floating-point Code via Unconstrained Programming (Extended Version). *arXiv preprint arXiv:1704.03394*, 2017.

[44] FuzzBench. Fuzzbench repository. https://github.com/google/fuzzer-test-suite, 2021. Accessed: 2021-08-06.

[45] GitHub. Github, public fuzzers. https://github.com/search?q=fuzzing&type=Repositories, 2021. Accessed: 2021-11-06.

[46] Github. Open-source repositories. https://github.com, 2021. Accessed: 2021-08-06.

[47] Patrice Godefroid. Micro execution. In *36th International Conference on Software Engineering*, pages 539–549, 2014.

[48] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 206–215, 2008.

[49] Jiong Gong, Yun Wang, Haihao Shen, Xu Deng, Wei Wang, and Xiangning Ma. FAST: Formal specification driven test harness generation. In *Tenth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMCODE2012)*, pages 33–42. IEEE, 2012.

[50] Peter Goodman and Alex Groce. DeepState: Symbolic unit testing for C and C++. In *NDSS Workshop on Binary Analysis Research*, 2018.

[51] Google. Google chromium security. https://www.chromium.org/Home/chromium-security/bugs, 2021. Accessed: 2021-11-06.

[52] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. " not my bug!" and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, pages 395–404, 2011.

[53] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, and Christophe Kruegel. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 135–150, 2019.

[54] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[55] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A Ground-Truth Fuzzing Benchmark. *ACM on Measurement and Analysis of Computing Systems*, 4(3):1–29, 2020. Publisher: ACM.

[56] Sean Heelan. *Greybox Automatic Exploit Generation for Heap Overflows in Language Interpreters*. PhD thesis, University of Oxford, 2020.

[57] Joseph L. Hellerstein. Configuring resource managers using model fuzzing: A case study of the. NET thread pool. In *2009 IFIP/IEEE International Symposium on Integrated Network Management*, pages 1–8. IEEE, 2009.

[58] Benjamin Robert Holland. *Computing homomorphic program invariants*. PhD thesis, Iowa State University, 2018.

[59] HongFuzz. Honggfuzz software. https://github.com/google/honggfuzz, 2021. Accessed: 2021-08-06.

[60] Wen-ling Huang, Niklas Krafczyk, Hoang M. Le, and Jan Peleska. Property-oriented Model-Based Testing With Fuzzing–Technical Report 09/2020–. Technical report, Department of Mathematics and Computer Science, University of Bremen, 2020.

[61] Vincenzo Iozzo. 0-knowledge fuzzing. *Black Hat USA*, 2010.

[62] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. Fuzzgen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287, 2020.

[63] Franjo Ivančić. SunDew: Systematic Automated Security Testing. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 3–3. IEEE, 2020.

[64] Pankaj Jalote. Coding and unit testing. In *A Concise Introduction to Software Engineering*, pages 1–43. Springer, 2008.

[65] Joonun Jang and Huy Kang Kim. FuzzBuilder: automated building greybox fuzzing environment for C/C++ library. In *35th Annual Computer Security Applications Conference*, pages 627–637, 2019.

[66] JavaDoc. Javadoc software. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html, 2021. Accessed: 2021-08-06.

[67] Jazzer. Jazzer software. https://github.com/CodeIntelligenceTesting/jazzer/, 2021. Accessed: 2021-08-06.

[68] Fan Jiang, Cen Zhang, and Shaoyin Cheng. FFFuzzer: Filter Your Fuzz to Get Accuracy, Efficiency and Schedulability. In *Australasian Conference on Information Security and Privacy*, pages 61–79. Springer, 2017.

[69] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. RULF: Rust Library Fuzzing via API Dependency Graph Traversal. *arXiv preprint arXiv:2104.12064*, 2021.

[70] Jinho Jung. *Practical Systems For Strengthening And Weakening Binary Analysis Frameworks*. PhD Thesis, Georgia Institute of Technology, 2021.

[71] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. In *2021 Annual Network and Distributed System Security Symposium (NDSS), Virtual*, 2021.

[72] Staffs Keele et al. Guidelines for performing systematic literature reviews in software engineering. Technical report, Citeseer, 2007.

[73] Matthew Kelly, Christoph Treude, and Alex Murray. A Case Study on Automated Fuzz Target Generation for Large Codebases. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6. IEEE, 2019.

[74] Atte Kettunen. Test harness for web browser fuzz testing. Master's thesis, University of Oulu, 2014.

[75] Fouad Nouri Khalaf. Web Protocol Fuzzing of the TLS/SSL Protocol with Focus on the OpenSSL Library. Bachelor's thesis, University of Queensland, 2020.

[76] Jee-Hyun Kim. Relevance of the cyclomatic complexity threshold for the web programming. *Journal of the Korea Society of Computer and Information*, 17(6):153–161, 2012.

[77] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[78] Balázs Kiss, Nikolai Kosmatov, Dillon Pariente, and Armand Puccetti. Combining static and dynamic analyses for vulnerability detection: Illustration on heartbleed. In *Haifa Verification Conference*, pages 39–50. Springer, 2015.

[79] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26, 2004.

[80] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. 2007.

[81] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[82] Zhifeng Lai. *Effective detection of atomic-set serializability violations in multithreaded programs.* Hong Kong University of Science and Technology (Hong Kong), 2010.

[83] Zhifeng Lai, Shing-Chi Cheung, and Wing Kwong Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 235–244, 2010.

[84] Hoang M. Le. KLUZZER: Whitebox Fuzzing on Top of LLVM. In *International Symposium on Automated Technology for Verification and Analysis*, pages 246–252. Springer, 2019.

[85] Leptonica. Leptonica software. http://www.leptonica.org/, 2021. Accessed: 2021-08-06.

[86] Randall J LeVeque, Ian M Mitchell, and Victoria Stodden. Reproducible research for scientific computing: Tools and strategies for changing the culture. *Computing in Science & Engineering*, 14(04):13–17, 2012.

[87] Nicole Lévy and G Smith. A language-independent approach to specification construction. *ACM SIGSOFT Software Engineering Notes*, 19(5):76–86, 1994.

[88] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 533–544, 2019.

[89] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018. Publisher: IEEE.

[90] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 562–566. IEEE, 2018.

[91] Jie Liang, Yu Jiang, Mingzhe Wang, Xun Jiao, Yuanliang Chen, Houbing Song, and Kim-Kwang Raymond Choo. Deepfuzzer: Accelerated deep greybox fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2019. Publisher: IEEE.

[92] Igor Lima, Jefferson Silva, Breno Miranda, Gustavo Pinto, and Marcelo d'Amorim. Exposing bugs in JavaScript engines through test transplantation and differential testing. *Software Quality Journal*, 29(1):129–158, 2021. Publisher: Springer.

[93] Antonio Lioy, Jan Tobias Müehlberg, Mathy Vanhoef, and Graziano Marallo. Identifying Software and Protocol Vulnerabilities in WPA2 Implementations through Fuzzing. Master's thesis, Politecnico di Torino, 2019.

[94] LLVM. libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2021. Accessed: 2020-12-04.

[95] LLVM. Llvm documentation. https://llvm.org/docs/LangRef.html, 2021. Accessed: 2021-08-06.

[96] Diomidis Spinellis Vassilios Karakoidas Panos Louridas. Comparative Language Fuzz Testing. *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2012.

[97] Dominik Maier and Lukas Seidel. JMPscare: Introspection for Binary-Only Fuzzing. In *Workshop on Binary Analysis Research (BAR)*, volume 2021, page 21, 2021.

[98] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE: baseband sanitized fuzzing through emulation. In *13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 122–132, 2020.

[99] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[100] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4): 308–320, 1976.

[101] Kenneth L. McMillan and Lenore D. Zuck. Formal specification and testing of QUIC. In *ACM Special Interest Group on Data Communication*, pages 227–240. 2019.

[102] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. Technical report, Defence SCience and Technology Organisation (Australia), 2012.

[103] Steven Mestdagh. Fuzzing Universal Plug and Play (UPnP). *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2019.

[104] Microsoft. Microsoft security development lifecycle, verification phase. https://www.microsoft.com/en-us/sdl/process/verification.aspx, 2021. Accessed: 2021-11-06.

[105] Microsoft. One fuzz. https://www.microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/, 2021. Accessed: 2021-08-06.

[106] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[107] Ryu Miyaki, Norihiro Yoshida, Natsuki Tsuzuki, Ryota Yamamoto, and Hiroaki Takada. Fuzz4B: a front-end to AFL not only for fuzzing experts. In *11th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 17–20, 2020.

[108] Mockingbird. Mockingbird framework. https://github.com/kcsl/Mockingbird, 2021. Accessed: 2021-08-06.

[109] Juho Myllylahti. Incorporating fuzz testing to unit testing regime. PhD Thesis, Master's thesis, University of Oulu, Oulu, Finland, 2013.

[110] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *2005 international workshop on Mining software repositories*, pages 1–5, 2005.

[111] Cuong Nguyen, Hiroaki Yoshida, Mukul Prasad, Indradeep Ghosh, and Koushik Sen. Generating Succinct Test Cases Using Don't Care Analysis. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[112] Hoang Lam Nguyen, Nebras Nassar, Timo Kehrer, and Lars Grunske. MoFuzz: A Fuzzer Suite for Testing Model-Driven Software Engineering Tools. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1103–1115. IEEE, 2020.

[113] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. DifFuzz: differential fuzzing for side-channel analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 176–187. IEEE, 2019.

[114] Yannic Noller. Differential program analysis with fuzzing and symbolic execution. In *33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 944–947, 2018.

[115] Yannic Noller, Corina S. Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. HyDiff: Hybrid differential software analysis. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1273–1285. IEEE, 2020.

[116] Timothy Nosco, Jared Ziegler, Zechariah Clark, Davy Marrero, Todd Finkler, Andrew Barbarello, and W. Michael Petullo. The Industrial Age of Hacking. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1129–1146, 2020.

[117] Joshua Ofoeda, Richard Boateng, and John Effah. Application programming interface (api) research: A review of the past to inform the future. *International Journal of Enterprise Information Systems (IJEIS)*, 15(3):76–95, 2019.

[118] Saahil Ognawala, Fabian Kilger, and Alexander Pretschner. Compositional fuzzing aided by targeted symbolic execution. *arXiv preprint arXiv:1903.02981*, 2019.

[119] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1481–1498, 2020.

[120] OpenCV. Opencv software. https://opencv.org/, 2021. Accessed: 2021-08-06.

[121] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[122] Rohan Padhye, Caroline Lemieux, and Koushik Sen. JQF: coverage-guided property-based testing in Java. In *28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 398–401, 2019.

[123] Annibale Panichella and Urko Rueda Molina. Java unit testing tool competition-fifth round. In *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*, pages 32–38. IEEE, 2017.

[124] Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 80–91. IEEE, 2018.

[125] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642. IEEE, 2020.

[126] Jan Peleska and Wen-ling Huang. Test Automation. 2021.

[127] Roger D Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011.

[128] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019. Publisher: IEEE.

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

[130] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. What is the vocabulary of flaky tests? In *17th International Conference on Mining Software Repositories*, pages 492–502, 2020.

[131] Fuzzing Project. Fuzzing project software. https://fuzzing-project.org/software.htm, 2021. Accessed: 2021-11-06.

[132] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. Towards making formal methods normal: meeting developers where they are. *arXiv preprint arXiv:2010.16345*, 2020.

[133] Per Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006.

[134] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *2010 IEEE Symposium on Security and Privacy*, pages 513–528. IEEE, 2010.

[135] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDss*, 2010.

[136] Google Scholar. Google scholar search engine. https://scholar.google.com.au/, 2021. Accessed: 2021-08-06.

[137] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[138] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 309–318, 2012.

[139] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE, 2016.

[140] Kostya Serebryany. Oss-fuzz-google's continuous fuzzing service for open source software. 2017.

[141] Muhammad Rabee Shaheen and Lydie Du Bousquet. Survey of source code metrics for evaluating testability of object oriented systems. 2010.

[142] Bhargava Shastry. *Compiler assisted vulnerability assessment*. PhD thesis, Technische Universität Berlin, 2018.

[143] Bhargava Shastry, Federico Maggi, Fabian Yamaguchi, Konrad Rieck, and Jean-Pierre Seifert. Static Exploration of Taint-Style Vulnerabilities Found by Fuzzing. In *WOOT*, 2017.

[144] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

[145] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 315–317, 2008.

[146] Maksim O Shudrak and Vyacheslav V Zolotarev. Improving fuzzing using software complexity metrics. In *ICISC 2015*, pages 246–261. Springer, 2015.

[147] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2541–2557, 2020.

[148] Saija Sorsa. Protocol fuzz testing as a part of secure software development life cycle. Master's thesis, Tampere University of Technology, 2018.

[149] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. FirmFuzz: automated IoT firmware introspection and analysis. In *2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, pages 15–21, 2019.

[150] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[151] Victoria C Stodden. Reproducible research: Addressing the need for data and code sharing in computational science. 2010.

[152] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–31, 2021.

[153] Fuzzer Test Suite. Google fuzzer-test-suite repository. https://github.com/google/fuzzer-test-suite/tree/51356066dc70c43c9da0ad98e887684a0394860f, 2021. Accessed: 2021-08-06.

[154] Haiyang Sun, Yudi Zheng, Lubomír Bulej, Stephen Kell, and Walter Binder. Analyzing Distributed Multi-platform Java and Android Applications with ShadowVM. In *Asian Symposium on Programming Languages and Systems*, pages 356–365. Springer, 2015.

[155] Yoshiki Takashima, Ruben Carlos Goncalves Martins, Limin Jia, and Corina Pasareanu. SyRust: Automatic Testing of Rust Libraries with Semantic-Aware Program Synthesis–Technical Report. 2021. Publisher: Carnegie Mellon University.

[156] Ahmed Tamrawi and Suresh Kothari. Projected control graph for computing relevant program behaviors. *Science of Computer Programming*, 163:93–114, 2018.

[157] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical software engineering*, 19(6):1665–1705, 2014.

[158] Scott Tenaglia and Perri Adams. Edge of the Art in Vulnerability Research. Technical report, Two Six Labs Arlington United States, 2020.

[159] Kashyap Thimmaraju, Bhargava Shastry, Tobias Fiebig, Felicitas Hetzelt, Jean-Pierre Seifert, Anja Feldmann, and Stefan Schmid. Reins to the Cloud: Compromising Cloud Systems via the Data Plane. *arXiv preprint arXiv:1610.08717*, 2016.

[160] Toby Tobkin. Implementation and testing of a blackbox and a whitebox fuzzer for file compression routines. *HIM 1990-2015. 1475.*, 2013.

[161] Dimitrios Tychalas and Michail Maniatakos. IFFSET: in-field fuzzing of industrial control systems using system emulation. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 662–665. IEEE, 2020.

[162] Dimitrios Tychalas, Hadjer Benkraouda, and Michail Maniatakos. ICSFuzz: Manipulating I/Os and Repurposing Binary Code to Enable Instrumented Fuzzing in ICS Control Applications. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[163] Frederick Ulrich. *Exploitability assessment with TEASER*. PhD thesis, Northeastern University Boston, 2017.

[164] Dion Van Dam. Analysing the signal protocol. Master's thesis, Radboud University, 2019.

[165] Eleanor Van Looy. Fuzzing Universal Plug and Play. Master's thesis, KU Leuven Faculty of Engineering, 2019.

[166] Mathy Vanhoef. WiFuzz: detecting and exploiting logical flaws in the Wi-Fi cryptographic handshake. In *Black Hat US Briefings, Location: Las Vegas, USA*, 2017.

[167] Juraj Vijtiuk. Deterring attackers by injecting unexploitable bugs. Master's thesis, University of Zagreb, 2009.

[168] James Walden. The impact of a major security event on an open source project: The case of openssl. In *17th International Conference on Mining Software Repositories*, pages 409–419, 2020.

[169] Zhiyuan Wan and Bo Zhou. Effective code coverage in compositional systematic dynamic testing. In *2011 6th IEEE Joint international information technology and artificial intelligence conference*, volume 1, pages 173–176. IEEE, 2011.

[170] Mingzhe Wang, Jie Liang, Chijin Zhou, Yuanliang Chen, Zhiyong Wu, and Yu Jiang. Industrial Oriented Evaluation of Fuzzing Techniques. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 306–317. IEEE, 2021.

[171] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 328–337. IEEE, 2021.

[172] Yan Wang, Wei Wu, Chao Zhang, Xinyu Xing, Xiaorui Gong, and Wei Zou. From proof-of-concept to exploitable. *Cybersecurity*, 2(1):1–25, 2019.

[173] Jane Webster and Richard T Watson. Analyzing the past to prepare for the future: Writing a literature review. *MIS quarterly*, pages xiii–xxiii, 2002.

[174] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. Singularity: Pattern fuzzing for worst case complexity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 213–223, 2018.

[175] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, pages 352–357, 1984.

[176] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *18th international conference on evaluation and assessment in software engineering*, pages 1–10, 2014.

[177] Affan Yasin, Rubia Fatima, Lijie Wen, Wasif Afzal, Muhammad Azhar, and Richard Torkar. On using grey literature and google scholar in systematic literature reviews in software engineering. *IEEE Access*, 8:36226–36243, 2020.

[178] Michal Zalewski. American fuzzy lop, 2014.

[179] Michal Zalewski. Technical “whitepaper” for afl-fuzz. *URl: http://lcamtuf. coredump. cx/afl/technical details. txt*, 2014.

[180] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book, 2019.

[181] K Zetter. A famed hacker is grading thousands of programs—and may revolutionize software in the process, 2016.

[182] Dazhi Zhang. *Detect Program Vulnerabilities Using Trace-based Security Testing*. PhD thesis, The University of Texas, 2011. Publisher: Computer Science & Engineering.

[183] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. IntelliGen: Automatic Driver Synthesis for Fuzz Testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 318–327. IEEE, 2021.

[184] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1105–1116, 2014.

[185] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. Efficient Fuzz Testing for Apache Spark Using Framework Abstraction. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 61–64. IEEE, 2021.

[186] Yu Zhang, Wenlong Feng, and Mengxing Huang. Automatic generation of high-coverage tests for RTL designs using software techniques and tools. In *2016 IEEE 11th Conference on Industrial Electronics and Applications (ICIEA)*, pages 856–861. IEEE, 2016.

[187] Yaowen Zheng, Zhanwei Song, Yuyan Sun, Kai Cheng, Hongsong Zhu, and Limin Sun. An Efficient Greybox Fuzzing Scheme for Linux-based IoT Programs Through Binary Static Analysis. In *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2019.

[188] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, Chengnian Sun, and Yu Jiang. VisFuzz: understanding and intervening fuzzing with interactive visualization. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1078–1081. IEEE, 2019.

[189] Milda Zizyte. *Better Robustness Testing for Autonomy Systems*. PhD Thesis, Carnegie Mellon University, 2020.

[190] Zotero. Zotero software. https://www.zotero.org/, 2021. Accessed: 2021-08-06.