On the Usability of Coverage-Based Fuzzing of C/C++ Programs

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von
Stephan Plöger
aus
Bielefeld

Bonn, August 2023

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen
Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn.

Erstgutachter: Prof. Dr. Matthew Smith
Zweitgutachter: Prof. Dr. Michael Meier

Tag der Promotion: 07.02.2024
Erscheinungsjahr: 2024

Acknowledgements

I would like to thank my advisor, Prof. Dr. Matthew Smith, who not only guided me along my journey but was also a calm anchor in stressful and hectic times. He always had a sympathetic ear for me and my projects. And I'm grateful for the opportunity he provided me to pursue my research.

I'm grateful to my family and friends, who gave me strength and endurance, and who always supported me on my adventure. Their cheerfulness and light-heartedness were a source of joy. It's nice to know your crowd is behind you.

My special thanks go to Anne, for her never-ending support and encouragement, especially in times of setbacks. She supported me at any point in time, and with the help of her confidence in me, I could overcome all challenges. She distracted me from work when necessary and cleared my head with her cheerful nature. It is just essential to have someone always by your side.

I would also like to thank my office buddy Mischa Meier, who gracefully accepted the "opportunity" to assist my research by coding and discussing every single interview of my studies that was ever transcribed. You are one of the true unsung heroes.

Also, I want to thank all my other colleagues at the Behavioral Security Group for their valuable input for my research, the chatting in the coffee kitchen, and just having a good time. I was very lucky to have such a supporting and benevolent working group.

Abstract

Even though the foundations for fuzzing were laid more than 30 years ago, it did not play a role in industry or academia for a long time. Interestingly, the popularity of fuzzing has risen for top-tier companies and academia in recent years. There are no firm findings on the reasons for this, but a continued increase in awareness of the need for secure software systems and access to more computing power may have played a role. In addition, fuzzing has proven to be capable of finding severe software vulnerabilities that slipped through the established security process. In academia, the focus on the topic of fuzzing lies heavily in improving performance, e.g., better execution speed or covering more code faster or making fuzzing applicable to more software and systems. However, examining the usability of fuzzing has yet to be touched.

A drawback of the current state of fuzzing is that especially small or medium-sized companies are not benefiting from it because they do not use it. The reasons for this still need to be explored.

Motivated by the lack of understanding about the usability of fuzzing and its potential negative influence on widespread adoption, I examine in this thesis the usability of fuzzers in the context of C and C++ programs by conducting four user studies.

In the first qualitative study with computer science (CS) students and Capture the Flag (CTF) players, I examined the usability of a fuzzer and a static code analysis tool, with static code analysis being a technique that is already very commonly used. This way, I found first insights into potential problems that hinder the adoption of fuzzing. The results revealed several usability issues, which I converted into recommendations.

I strengthened the qualitative findings of my first study with a quantitative comparison of the two most popular fuzzers, AFL and libFuzzer, in a second user study with CS students. Large parts of the original recommendations could be confirmed and supplemented. Moreover, I could show that AFL performed better overall and was rated higher than libFuzzer.

To be able to make a more generalizable statement on the findings, I replicated my second study, but this time with freelance developers. With this study, I provided insights into the methodological implications of substituting freelancers with students in fuzzing studies and solidified the quantitative results for both fuzzers.

In my fourth study, I investigated whether developers should fuzz their self-written code or code written by others. This was motivated by existing examples in related fields for and against analyzing someone's own code. In the results, I could not find any evidence that a practical difference exists between fuzzing self-written code and the code of others.

Based on the results of the four studies, I additionally give recommendations to improve the usability of fuzzers.

Contents

Chapter 1

Introduction

Adding security to the software development life cycle has been an ongoing process for many years. The need for this ultimately arises from an ever-increasing threat situation. In the beginning, security experts like software testers were working on the security of the software alone. In recent years, however, a distribution of responsibility has begun so that more and more parties, such as developers, are involved. This is due to a change in the distribution of tasks and duties but also because smaller companies feel obliged to invest in the security of their software. Security experts are a rare good, and therefore small or even medium-sized companies frequently do not have the capacity to provide a designated security expert for a software project. Consequently, developers have to take over this task.

There exist many different techniques and approaches to adapt security in the software development life cycle. One of them is security testing, which is the field where the topic of my thesis lies. Security testing is a science in itself. A plethora of tools and techniques exist, which are built for different parties. From a developer's point of view, static code analysis [34], also known as static application security testing (SAST), has been a constant for many years. Static code analysis aims to find software bugs by analyzing the program's source code. It is used by very many companies of different sizes – ranging from small to very large – including Google [164], and there is a variety of commercial [44, 125, 29, 41, 184] as well as open-source [69, 147, 39] SAST tools on the market. Tools exist for nearly any programming language that is used in production. Static code analysis is also a recurring theme in academia, where not only the capabilities and speed in finding bugs have been tested and improved but where also, albeit to a lesser extent, the usability of such tools has been investigated in both interviews and user studies [172, 173, 174, 99, 36, 183].

In recent years, another type of application security testing has emerged: fuzzing. Fuzzing is a dynamic analysis technique, which means that the program under test is run to find bugs. It thus forms a counterpart to static code analysis.

In industry, an ambiguity about fuzzing exists. On the one hand, it is already extensively and effectively used by large companies such as Microsoft [127], Cisco [37], and Google – Google found more than 40,000 bugs in 650 open source projects [168], 26,000 of them in Chrome alone [26]. On the other hand, small and medium-sized companies have yet to discover fuzzing for themselves, and the reasons for that have not been examined so far. Moreover, fuzzers are still in their infancy. The market for commercial fuzzers is very small, if not even non-existent. Even in the open-source area, the choice is small [159, 88, 117, 4, 5], and existing fuzzers are typically for programs written in C or C++. Although, companies and developers are actively working on making fuzzing available for more programming languages [76, 163].

There is also a lot of interest in fuzzing in the scientific community. Many efforts have been made to further improve the speed and recognition rate and make fuzzing accessible to other systems, such as other kernels [35, 43]. However, an ambiguity exists here as well. Research into the usability of fuzzers has so far remained entirely untouched.

1.1 Research Contribution

Motivated by the lack of understanding about the usability of fuzzing and its potential negative influence on widespread adoption, I examined in this thesis the usability of fuzzers in the context of C and C++ software from different points of view.

At first, I qualitatively evaluated the usability of the prevalent software testing technique, static code analysis, and the challenger fuzzing in a within-subjects user study with computer science students and Capture the Flag (CTF) players [27]. I recruited students as the next best alternative to professional software developers because recruiting these is very difficult. Reasons include that the group of professional software developers is, on the one hand, relatively small and, on the other hand, highly educated, well-paid, and oftentimes challenged and under time pressure in their full-time jobs [1, 2, 171, 107]. To complicate matters, working with a fuzzer takes several hours, which makes recruitment even more difficult. Moreover, in related fields of study, such as user studies with programming tasks, others showed that computer science students can successfully substitute software developers with a high chance of receiving similar results in relative terms [136, 137, 138, 3, 2].

In the study, the computer science students worked on two easy or hard tasks, which were randomly assigned. Each participant used once the open-source static code analyzer Clang Static Analyzer [39] (CSA) and once the open-source fuzzer libFuzzer [117]. The study design for the CTF players was the same; however, as they were considered hackers with broader knowledge about the topic, they only worked with the hard tasks. The results indicate that the CSA had, based on the definitions of Nielsen [144], a good usability, which means it was easy to use, but had a bad utility, which means a bad effectiveness and efficiency. In contrast, the participants rated the usability of fuzzing as very bad but perceived the utility as better. Therefore, I saw a lot of potential for the fuzzer libFuzzer and proposed recommendations to improve its usability.

Unfortunately, many participants dropped out of the study because it was too challenging for them or they got stuck fairly early in the fuzzing process so that only very little information on the usability in the later steps could be gathered.

As a consequence, I conducted a second user study with computer science students to, on the one hand, gather more information on the usability of fuzzing, especially in the later steps, and also compare the usability of the two most popular fuzzers AFL [4] and libFuzzer. To make sure that participants reached deeper into the fuzzing process, I guided them through the fuzzing process with subtasks in the task description and also implemented a support system where they could get help if they got stuck working on the task. With the help of the results, I confirmed some of the recommendations proposed in my first study and could also extend them, especially for the later steps in the fuzzing process. Moreover, I showed that the CS students scored significantly better results with AFL than with libFuzzer. Although, despite the introduction of guiding subtasks and a support system, the dropout rate was still very high.

So far, both studies have examined the usability of fuzzing considering students and CTF players, but a generalizability of the results to professional software developers cannot be assumed without further ado. To tackle this topic, I conducted a third study, which is a replication of the comparison of AFL and libFuzzer, but this time with freelance developers instead of CS students. This way, I provided first insights on the substitution of developers with CS students and solidified the quantitative results for both fuzzers. The results of the study were similar to those of the previous study. The participants performed better using AFL than libFuzzer in terms of success and fuzzing score. Moreover, the usability of AFL was rated higher. These results indicate that a substitution is possible. Interestingly, the freelancers performed slightly worse than the CS students overall.

In my last study, CS students fuzzed their own code or code written by others. This way, I gained first insights on who in a company should take on the task of fuzzing. As I already briefly touched on, small and medium-sized companies often do not have fuzzing or security experts in their ranks. Therefore, software developers do the testing. It is common for developers not to test their own code but the code of others, as it is standard practice for code reviewing. The idea behind that is simple. A fresh set of eyes that has less of an emotional connection to the code and the work behind it can find bugs better than developers that examine their code themselves. However, to the best of my knowledge, there is no empirical evidence that this actually helps to find more bugs. As a counter-example, unit tests are typically created by the developer of the code instead of a coworker. So the question I tackled with my study was whether it is more helpful to fuzz your self-written code or that written by another person. The results did not show any evidence that a practical difference exists between fuzzing someone's own code or the code of others.

1.2 Thesis Structure

My thesis is structured as follows:

Chapter 1 In the first chapter, I introduce and motivate my thesis topic and present its structure.

Chapter 2 I embed my work in the scientific context in Chapter 2. The content of this chapter was partially published in my work "A Qualitative Usability Evaluation of the Clang Static Analyzer and libFuzzer with CS Students and CTF Players" [156].

Chapter 3 In the third chapter, I give an introduction to fuzzing itself. I explain the origin and briefly describe the idea of the type of fuzzing relevant to this thesis. Moreover, I present an easy and a hard example of fuzzing tasks and their solutions for both relevant fuzzers to get a better understanding of fuzzing.

Chapter 4 Chapter 4 describes a qualitative usability evaluation of the static code analyzer Clang Static Analyzer and the fuzzer libFuzzer with CS Students and CTF Players. The content was previously published as parts of the publication "A Qualitative Usability Evaluation of the Clang Static Analyzer and libFuzzer with CS Students and CTF Players" by Stephan Plöger, Mischa Meier, and Matthew Smith, presented at the Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021).

Chapter 5 Chapter 5 describes the comparison of AFL and libFuzzer with CS students. The content is at the time of writing this thesis under review as part of the paper "A Usability Evaluation of AFL and libFuzzer with CS Students".

Chapter 6 Chapter 6 describes the replication of the AFL and libFuzzer study but this time with freelance developers.

Chapter 7 Chapter 7 describes the comparison of developers fuzzing their own code or code written by others.

Chapter 8 In Chapter 8, I summarize my studies presented in Chapters 4 to 7. Based on my findings, I give recommendations on how fuzzers can be improved to be more usable.

In the disclaimers at the beginning of Chapters 4 to 7, I provide details on my co-authors' and my contributions to that particular work. To express my gratitude for my colleagues' and co-authors' help in the studies, the personal pronoun "we" is used in place of "I". Their support was an integral part of what made the studies possible.

Chapter 2

Related Work

In this chapter, I would like to present previous work on usability studies on fuzzing and static code analysis to embed the studies presented in Chapters 4 to 7 into the scientific context. I start by discussing related work on the usability of static code analysis. Second, I present publications tackling the usability of fuzzing. I then address related work for the substitution of professional developers with freelancers or computer science students before presenting studies with freelance developers. And last but not least, I discuss related work on user studies that work with self-written code.

2.1 Usability of Static Code Analysis

The works on the usability of static code analysis, also known as static application security testing, can be loosely divided into five categories: data analyses [94], interview and survey studies [99, 183, 60], task-based user studies [172, 174, 173, 177, 180, 10], and analyses of practical usage [36, 164, 8, 93, 12].

Data analyses In 2019, Imtiaz et al. [94] presented an analysis of 280 Stack Overflow questions asked by developers about static analysis tool alerts. They found that the top six types of questions constituted more than 90 % of the questions asked. The most prominent ones were about ignoring and filtering alerts and false positive validation.

Interview and survey studies In 2013, Johnson and Song [99] conducted 20 interviews about static code analysis with 20 developers. They found that most participants felt that using static analysis tools is beneficial but that the high number of false positives and the presentation of the bugs were demotivating. Moreover, the participants expressed the importance of workflow integration. Participants also stated that the tools should make suggestions for a fix.

In 2016, Christakis and Bird [36] conducted a survey at Microsoft to get more insights into the use of static code analysis. They set the focus on the barriers to using static analysis, the functionality that the developers desire, and the non-functional characteristics that a static analyzer should have. They also found that false-positive rates were the main factor leading developers to stop using the analyzer. Developers were willing to guide the analyzer on where to focus with their input and desired customizability and the option to prioritize warnings.

Vassallo et al. [183] confirmed those findings in 2018. They surveyed 42 developers and interviewed 11 industrial experts to shed light on the influences of the development context on usability issues. Besides other things, they found that developers mainly used DAST tools in three different contexts, local environment, code review, and continuous integration, and that many developers paid attention to different warning categories depending on the development context.

Task-based user studies Smith et al. [172] conducted a heuristic walkthrough and a user study on the usability of four static analysis tools. They used Find Security Bugs and an anonymized commercial tool for Java, RIPS for PHP, and Flawfinder for C. They identified several issues ranging from problems with the inaccuracy of the analysis, over workflow integration, to features that do not scale. They also conducted a think-aloud study in 2013 with five professional software developers and five students who had contributed to a security-critical Java medical records software system [174, 173]. They aimed to study users' needs while assessing security vulnerabilities in software code. The participants worked on four tasks for a maximum of one hour in a lab. However, participants were only asked to examine the reports of the static analysis tool and fix potential bugs, not to run the tool itself. Based on their findings, they gave recommendations for the design of static analysis tools. Their main suggestion was that tools should help developers search for relevant web resources.

In 2021, Tahaei et al. [177] ran an online experiment with 132 participants on the helpfulness of SAT guidance. They recruited their participants from GitHub, an online platform that provides participants for studies, and snowball sampling. Participants were shown four vulnerable Java code samples with the guidance of one of two SAT tools or no guidance at all. The participants were asked to provide the appropriate fix to the vulnerable code sample. They found that the SAT tools' guidance led to slightly improved code correction answers and to more detailed open-ended answers overall.

Analyses of practical usage In 2015, Sadowski et al. [164] conducted an in-situ evaluation of their static analysis tool Tricorder at Google. They presented their goals for the system and their lessons learned, such as empowering users to contribute and customizing the DAST tool based on the project and not based on the user.

A case study on alert occurrences and triage of those alerts on five open-source projects with Coverity was conducted by Imtiaz et al. [93] in 2019. They found that the fixes of Coverity are generally low in complexity but that developers take 96 days (median) to fix these. Moreover, their data suggest that the severity of the alert and fix complexity may correlate.

Miscellaneous In addition to this, Nachtigall et al. [131] surveyed research publications on static code analysis in the past decade, summarized the usability challenges, and found that the reported usability challenges fell into six categories:: understandable warning messages, fix support, false positives, user feedback, workflow integration, and specialized user interface.

Also, Nachtigall et al. [132] provided a comprehensive view of the usability of SAST tools by extracting usability measuring criteria from a detailed literature review and evaluating current SAST tools based on them. Even though they used a complementary approach to the aforementioned interview and survey, task-based studies, and use cases, they could confirm many previous findings and suspicions. They found that many tools did not address the areas of explaining warning messages, fix support, handling of false positives, consideration of user feedback, integration into the developer's workflow, and supporting user interfaces, especially not CLI-based ones. They showed that the usability of SAST tools still needed improvement.

Furthermore, a comparison of open-source static analysis tools for C/C++ code was conducted by Arusoaie et al. [6] in 2017. They compared 11 analysis tools on the Toyota ITC test suite [169]. They ranked them by productivity, balancing the detection rate with the false-positive rate to compensate for a high false-positive rate. The top three performers were clang [38], Frama-C [69], and OCLint [147].

2.2 Usability of Fuzzing

While the usability of static code analysis tools has been studied in various ways in recent years [172, 174, 173, 99], to my knowledge, there is very little work on the usability of fuzzers.

In 2021 Li et al. [115] constructed a benchmark for fuzzers consisting of 20 real-world programs. They also tested the usability of 35 fuzzers by manually building and testing each of the fuzzers. They found more than 15 flaws in this process. They suggest ensuring that the documentation is complete and correct, the fuzzer can successfully be installed, the fuzzer is robust, and the results are reproducible.

In 2020 Nosco et al. [146] performed a user study to find an effective process for finding bugs. In their study, 12 participants from the US Cyber Command personnel with different skill levels ranging from apprentice over journeyman to master worked in two teams. Their main tool to hunt for bugs was AFL, in which they received short training. They scored each participant based on the number of working harnesses, the number of bugs found, and the number of bugs reproduced. Unfortunately, due to their different focus, those numbers are not shared.

2.3 Ecological Validity of Student Samples

As it is very challenging to recruit professional developers for studies due to lack of time and very high costs [1, 2, 171, 107], students as substitutes are of interest. Although it is assumed that students can substitute professional developers for software engineering, this still needs to be determined regarding software security and IT security [107]. Nevertheless, indications for that can be found. Naiakshina et al. [136] compared password storage of students, freelancers, and professional software developers in German companies. They concluded that CS students could be used in comparative studies in their case even though they performed overall better than CS students because the effect of the dependent and independent variables held nonetheless. In 2017 Acar et al. [3] conducted a study with GitHub users on security-relevant programming tasks. The results showed that "neither student nor professional status was a significant factor for functionality, security, or security perception". They also conducted a user study in 2016 on the impact of information sources on code security, where participants had to complete four programming tasks [2]. The results did not show a difference in security for professionals and non-professionals.

Moreover, Tahaei and Vaniea [176] published their study on recruiting participants with programming skills. They recommended using CS mailing lists to recruit participants for studies where programming skills are required but also suggested collaborating with other universities to reduce validity issues. Besides that, Kaur et al. [103] compared six software developer samples to tackle the question of where to recruit for security development studies. They found that participants of all six recruiting platforms, including the student sample, report rich general software development and security experience, skills, and knowledge.

2.4 Studies with Freelancers

Gutfleisch et al. [84] published a study in 2022 where they explored how developers and other decision-makers encounter and deal with security and usability during the software development process in their companies. They conducted 25 semi-structured interviews. Of the 25 participants, they recruited ten from the freelancer platform Upwork. They highlight a need for more understanding of usable security and overall awareness, as well as strengthening communication between security and usability experts.

In 2013, Yamashita and Moonen [201, 200] conducted an online survey with 85 freelancers from freelancer.com to understand better the level of knowledge on code smells. They found that a considerably large portion of the participants, 32%, stated that they did not know about code smells. The majority of the participants were moderately concerned with respect to the perceived criticality of code smells. Moreover, the freelancers stated that they used technical blogs, programmer forums, colleagues, and industry seminars as their main sources of information.

Bau et al. [11] published a tech report in 2013 where they conducted a study about the security evaluation of 27 web applications developed by developers from 19 Silicon Valley startups and 8 freelancers. They found a disconnect between developer security knowledge and actual measurable vulnerability rate. Moreover, in their study, freelancers were significantly more prone to injection vulnerabilities than startups. Although, they theorized that the difference might be more attributable to motivation than education.

Naiakshina et al. [136] conducted a study with freelancers from freelancer.com in 2019 where they replicated a former study of them where participants had to program a password storage back-end for a web page of a startup. The 43 freelancers produced similar results as the students. In contrast to Bau et al., they found the freelancers very dependable.

Danilova et al. [48, 51] published two papers in 2020 and 2021 about studies with freelancers. In the first paper, they replicated the study of Naiakshina et al. about password storage. They altered the deception of the original study by announcing the project as a study and not as a real project. They found that the deception did not have a large effect and that open recruitment without deception was a viable recruitment method.

In the second paper, they conducted a study on code reviewing as a methodology for Online Security Studies. For that purpose, they recruited 44 developers from freelancer.com. The participants had to review an insecure password storage code snippet. They investigated how participants behaved in a code-reviewing study. Not even one-third reported the security issue, and almost all the participants who reported an issue were prompted for security.

2.5 User Studies - Working with Self-Written Code

The field of user studies where participants worked with their self-written code is rather scarce. As Rojas et al. [161] already pointed out in 2015, studies with testers testing their self-written code instead of code written by others were nonexistent. Rojas et al. studied the usefulness of automated unit test generation during the software development process. In their study, participants had to write two classes based on JavaDoc specifications and had to write unit tests that achieved the highest possible branch coverage. They had to manually write the unit test for one class and for the other class with their unit test generation tool EvoSuite.

To the best of our knowledge, the landscape of own vs. other software testing has mostly stayed the same since then. It is centered around software comprehension and recognition and only scratches testing or analyzing and debugging code.

A Secure Programming Clinic was developed by Dark et al. [53] in 2016. The clinic was tested in a programming class at a US university. It included graded programming homework, which could be resubmitted after visiting the clinic. They showed that using the clinic significantly increases robust programming compared to not using it.

In 2021 Lehtinen et al. [112] conducted a study with CS students from an introductory course in programming about the students' understanding of their own written code. They used automatically generated questions developed in a previous paper to measure the understanding of their own code [113]. They found a similar ratio of students failing to explain their own code as previous studies with non-self-written code have shown.

Krüger et al. [108] investigated *forgetting* in software development with a user study with 60 open-source developers in 2018. Participants had to fill out a survey which included a pre-submitted file of self-written code the questions were designed around. Besides other things, they found a need to understand better how developers track their code.

Chapter 3

Background

In this chapter, I would like to give a short overview of static code analysis and fuzzing. Both testing techniques will be presented with a brief introduction to their history as well as with an example of their usage. The example is provided to give a sense of usage and usability.

In addition, for fuzzing, I will also give an overview of how it works since the topic of fuzzing and its usability is the main focus of the thesis.

3.1 Static Code Analysis

As for many topics, determining the exact origin can be challenging. This is also true for static code analysis. Nevertheless, the program checker Lint[100] from Johnson and Murray published in 1978, is one of the first analysis tools that left a lasting impression. It was the namesake of lint programs, or linters, which highlight programming errors and bugs in code.

Other milestones in the development of static code analysis tools are difficult to identify. It is perhaps more appropriate to speak of a steady development. Altogether an upswing of the topic could be determined at the beginning of 2000. At this time, some companies [44, 29, 184], which are well-known today, were founded, which drive products out in the range of the static code analysis.

From that point onwards, static code analysis became more and more an integral part of the development and operations lifecycle. This was also reinforced by the fact that it was introduced to many coding standards, like NIST [13], BSI [170], or the UK Defense Standard [64], over the years.

I do not want to go into detail about individual static code analysis techniques, as this would exceed the scope of this thesis. However, I would like to present an example to give an impression of how a static code analysis tool can be used and to give insights into its usability.

3.1.1 Example of a Static Code Analysis

As an example of a static code analysis, I would like to show the process for the Clang Static Analyzer [39] used on Tesseract [178], an open-source optical character recognition program and library. I took this example from a paper my

```
scan-build ./configure

scan-build ./make
```

LISTING 3.1: Fuzz Target of AFL for tomlc99

```

scan-view *file*
```

LISTING 3.2: Fuzz Target of AFL for tomlc99

colleagues and I previously published, titled "Qualitative Usability Evaluation of the Clang Static Analyzer and libFuzzer with CS Students and CTF Players".

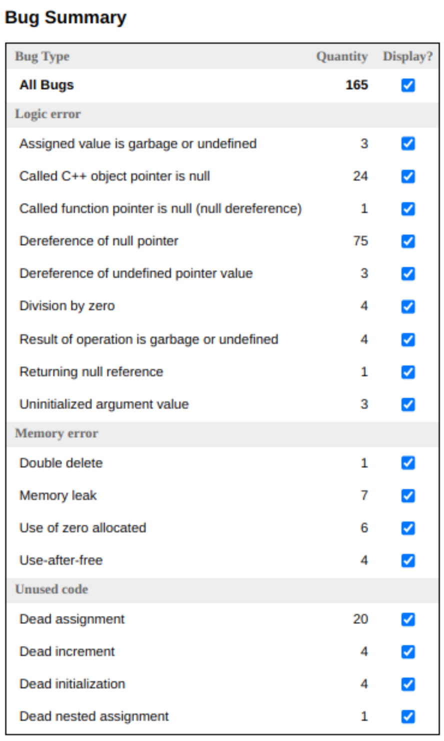
Doing a static code analysis with the Clang Static Analyzer can be divided into two steps.

Step 1: Building the target program with the analyzer In the first step, the target program, in this case, Tesseract, needs to be build with the analyzer, Clang Static Analyzer. This is done by combining the two building commands of Tesseract, `configure` and `make`, with the `scan build` command of the analyzer. The commands are shown in Listing 3.1. Combining the commands results in the interposing of the analyzer on the project's build process. This way, the analyzer analyzes every source file that is compiled by the compiler during the project build. However, if a source file is not part of the particular building process, it is thus also not analyzed. The addition of the static code analysis leads to an increase in the compile time. However, even for large projects like Tesseract, the build is typically completed in the range of minutes.

The Clang Static Analyzer consists of a plethora of options, including so-called checkers that can be turned on or off, which scan the code for specific types of bugs. Those include, e.g., checks for divisions by zero, dereferencing of null pointers, double frees and use-after-frees, and checks for index out-of-bounds. When the analyzer is run without any options, a basic set of checkers is used by default.

Step 2: Viewing and analyzing the findings After the target program is built and the analyses are finished, the results, including the findings, can be viewed by opening the created HTML file. This is done by using the `scan-view` command, as shown in Listing 3.2, is provided.

On the HTML page, a detailed summary of all findings, called bugs, is shown. The bug summary for Tesseract can be seen in Figure 3.1. It can be seen that with the basic set of checkers, the Clang Static Analyzer reports 165 findings. Of these 165 findings, to the best of my knowledge, only 1 is a true positive, resulting in a true positive rate of .06%. This directly highlights one of the biggest issues of static code analyzers, the too-high false-positive rate.



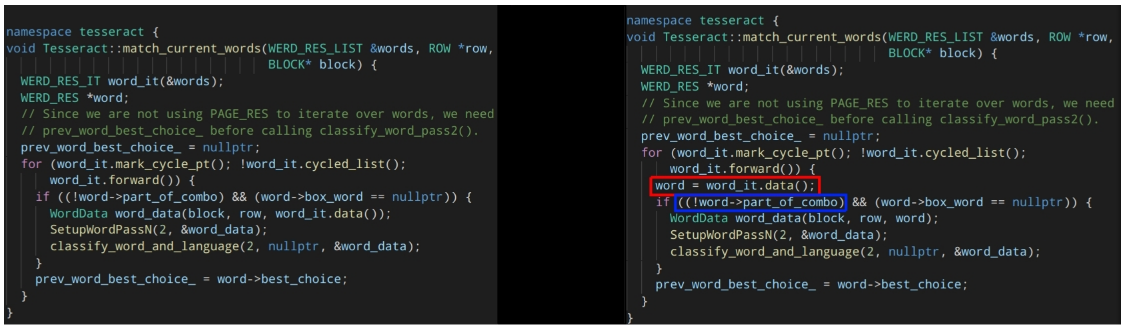
Bug Summary

Bug Type	Quantity	Display?
All Bugs	**165**	✓
Logic error		
Assigned value is garbage or undefined	3	✓
Called C++ object pointer is null	24	✓
Called function pointer is null (null dereference)	1	✓
Dereference of null pointer	75	✓
Dereference of undefined pointer value	3	✓
Division by zero	4	✓
Result of operation is garbage or undefined	4	✓
Returning null reference	1	✓
Uninitialized argument value	3	✓
Memory error		
Double delete	1	✓
Memory leak	7	✓
Use of zero allocated	6	✓
Use-after-free	4	✓
Unused code		
Dead assignment	20	✓
Dead increment	4	✓
Dead initialization	4	✓
Dead nested assignment	1	✓

FIGURE 3.1: Clang Static Analyzer - run summary

Detached from this issue, the Clang Static Analyzer allows, as all static code analysis tools, to take a detailed look at every finding. For the true positive, these details are shown in Figure 3.2.

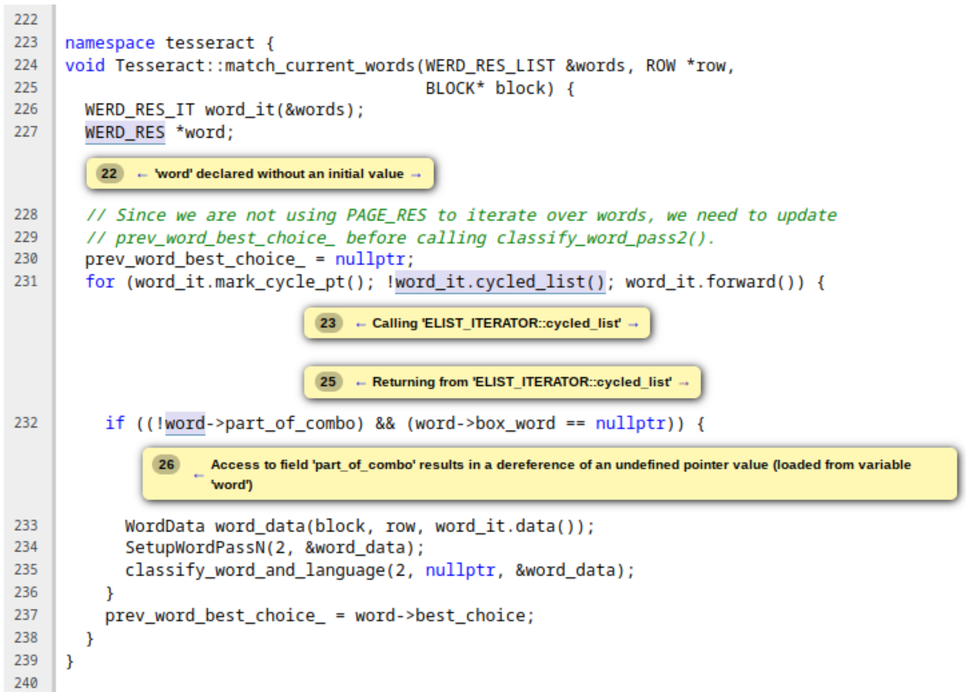
The analyzer assesses the situation correctly by deciding that accessing the field `word->part_of_combo` results in a dereference of an undefined pointer value. The analyzer not only points out the issue but also provides a path to its conclusion, starting in line 227, where it is stated that the variable `word` is declared. The reason for that is that the definition of the variable `word` was deleted by me from the original code to create this bug. The difference between the original code and the altered one can be seen in Figure 3.3.



```
namespace tesseract {
void Tesseract::match_current_words(WERD_RES_LIST &words, ROW *row,
                                    BLOCK* block) {
  WERD_RES_IT word_it(&words);
  WERD_RES *word;
  // Since we are not using PAGE_RES to iterate over words, we need
  // prev_word_best_choice_ before calling classify_word_pass2().
  prev_word_best_choice_ = nullptr;
  for (word_it.mark_cycle_pt(); !word_it.cycled_list();
      word_it.forward()) {
    if ((!word->part_of_combo) && (word->box_word == nullptr)) {
      WordData word_data(block, row, word_it.data());
      SetupWordPassN(2, &word_data);
      classify_word_and_language(2, nullptr, &word_data);
    }
    prev_word_best_choice_ = word->best_choice;
  }
}
```

```
namespace tesseract {
void Tesseract::match_current_words(WERD_RES_LIST &words, ROW *row,
                                    BLOCK* block) {
  WERD_RES_IT word_it(&words);
  WERD_RES *word;
  // Since we are not using PAGE_RES to iterate over words, we need
  // prev_word_best_choice_ before calling classify_word_pass2().
  prev_word_best_choice_ = nullptr;
  for (word_it.mark_cycle_pt(); !word_it.cycled_list();
      word_it.forward()) {
    word = word_it.data();
    if ((!word->part_of_combo) && (word->box_word == nullptr)) {
      WordData word_data(block, row, word);
      SetupWordPassN(2, &word_data);
      classify_word_and_language(2, nullptr, &word_data);
    }
    prev_word_best_choice_ = word->best_choice;
  }
}
```

FIGURE 3.3: Code comparison altered and original code



```
namespace tesseract {
void Tesseract::match_current_words(WERD_RES_LIST &words, ROW *row,
                                    BLOCK* block) {
  WERD_RES_IT word_it(&words);
  WERD_RES *word;
```
22 ← 'word' declared without an initial value →
```
  // Since we are not using PAGE_RES to iterate over words, we need to update
  // prev_word_best_choice_ before calling classify_word_pass2().
  prev_word_best_choice_ = nullptr;
  for (word_it.mark_cycle_pt(); !word_it.cycled_list(); word_it.forward()) {
```
23 ← Calling 'ELIST_ITERATOR::cycled_list' →

25 ← Returning from 'ELIST_ITERATOR::cycled_list' →
```
    if ((!word->part_of_combo) && (word->box_word == nullptr)) {
```
26 ← Access to field 'part_of_combo' results in a dereference of an undefined pointer value (loaded from variable 'word')
```
      WordData word_data(block, row, word_it.data());
      SetupWordPassN(2, &word_data);
      classify_word_and_language(2, nullptr, &word_data);
    }
    prev_word_best_choice_ = word->best_choice;
  }
}
```

FIGURE 3.2: Clang Static Analyzer - finding details

A fix for the bug would therefore be to reinstate the definition of the variable.

Overall, this example can be considered a complex case. Not because it is difficult to get the analysis running but rather due to the large amount of findings and the low number of actual true positives.

3.2 Fuzzing

> It started on a dark and stormy night. One of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. It was a race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash. [...] It is reasonable to expect that basic utilities should not crash ("core dump"); on receiving unusual input, they might exit with minimal error messages, but they should not crash.

This is a part of the introduction of a publication by Miller et al. [128], which was the foundation of the term fuzzing and fuzzing itself. In their publication, they presented a program called *fuzz*, which did similar things as the rain described in the introduction. *fuzz* generates random characters. These random characters can then be used to give them to the program under test as input. In the process, the target program is executed with new random inputs over and over again. If the target program crashes, a bug is found. This is the process of fuzzing.

In the early years of fuzzing, from the 1990s till about 2007, research on fuzzing was rather sparse. However, from 2007 onward, the number of publications has drastically increased with a strong polynomial growth [116]. Böhme et al. [16] attributed this to a tremendous need. Everyday life and nearly every business are highly dependent on computer systems. Consequently, security vulnerabilities can have major consequences. This realization has also led to the development of a different mindset where software security is becoming increasingly important. Companies are also trying new ways to make the software even more secure. For example, Google and Microsoft have therefore integrated fuzzing into their general workflow with success. Moreover, companies have installed bug bounty programs that pay good money for finding and reporting bugs [23, 24, 25]. This incentivizes developers and enthusiasts to look for vulnerabilities and develop tools to support them, benefiting the fuzzers. Good fuzzers are freely available and open source.

3.2.1 How does Fuzzing Work?

There are many different ways in which fuzzing can be performed and applied. These include approaches that fall into the categories of black-box, white-box, and gray-box testing [16].

Since this thesis is focused on studies with software in which the source code is available to the tester and written for common servers and home computers, I will only give an overview of the fuzzing technique most applicable to that situation. Moreover, as the studies addressed in this thesis are conducted with programs under test written in C and C++, I will not get into detail about fuzzing programs written in other programming languages.

The type of fuzzing I'm presenting in this thesis is usually referred to as feedback-based fuzzing. The basic idea of feedback-based fuzzing can be explained with the help of Figure 3.4. A fuzzer feeds an input to a system or program under test (PUT), which can also be called a target program (TPr), monitors the behavior of the program, which is done by afore-applied instrumentation and uses that information to generate a new input to feed it to the TPr again. A fuzzer has successfully found a bug when it provides an input to the TPr that causes the TPr to behave so that the operating system forbids the action, which usually results in the TPr crashing. A common example is an attempt to read from or write to a memory address that does not belong to the program under test.

For example, a TPr could be a PDF viewer, and the fuzzer could start the TPr with this thesis. Normally, the PDF viewer will take the PDF, open it by reading its content, and then display it. This process, however, is monitored so that the fuzzer can retrieve information on which code of the PDF viewer was used to handle the input, which parts of the code were not used, and how

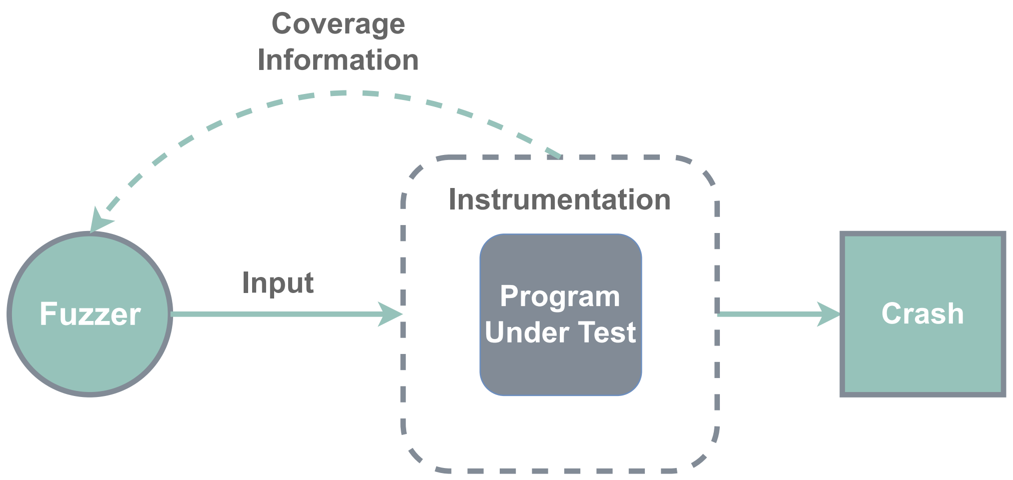


FIGURE 3.4: Feedback-based Fuzzing

the PDF viewer behaved overall. This monitoring is established when the PDF viewer itself is created. That means that when the PDF viewer is built out of its source code, additional instructions are added to the PDF viewer to provide the information to the fuzzer. The addition of additional instructions to provide information to other programs is called *instrumentation* in this context. Furthermore, the information itself is called *coverage information*. It tells the fuzzer which and how much code of the TPr is reached, respectively covered, by the input.

Moreover, depending on the fuzzer used, it can be necessary to write a snippet of code in which the parts, respectively, the functions of the TPr, to test the TPr are called. This snippet is called a *fuzz target* (FT).

In the end, when a crash is encountered, the root cause of the crash has to be determined.

Overall, fuzzing can therefore be divided into the following steps:

1. *Find a suitable function to fuzz.*
2. *Write your fuzz target.*
3. *Compile and instrument the target program.*
4. *Compile the fuzz target.*
5. *Run the fuzzer and interpret the output.*

3.2.2 Fuzzing Examples

In the following, I would like to give a short impression of the fuzzing process with two examples for the fuzzers AFL and libFuzzer. The first example is rather simple, while the second is more complex. Both examples are structured based on the five fuzzing steps mentioned above.

```
#include <unistd.h>
#include <string.h>

#include "toml.h"


int main() {
  char buf[100];

  while (__AFL_LOOP(1000)) {
    char errbuf[200];

    memset(buf, 0, 100);
    read(0, buf, 100);

    toml_table_t* tab = toml_parse(buf, errbuf, 200);

    toml_free(tab);
  }

  return 0;
}
```

LISTING 3.3: Fuzz Target of AFL for tomlc99

AFL and libroxml

I took this example from the previously published paper "A Usability Evaluation of AFL and libFuzzer with CS Students". The fuzzer AFL is used to fuzz the target program libroxml. This is arguably the easiest example possible for a fuzzing task.

Step 1: Find a suitable function to fuzz To fuzz tomlc99, I choose the toml_parse function. It is the very first function that needs to be used to interact with the library, and from that point on, a lot of the library can be covered.

Step 2: Write the fuzz target The fuzz target with the signature while-loop for using the persistent mode is shown in Listing 3.3. It reads the input from stdin, writes the input into a buffer, and calls toml_parse with this buffer. Moreover, an error buffer is created and passed to the function. In the end, the resulting toml table is freed.

Step 3: Build and instrument the target program The target program is built and instrumented with the command presented in Listing 3.4. The make command is invoked with the compiler set to the AFL compiler.

Step 4: Compile the fuzz target Then the fuzz target is built with the command also shown in Listing 3.4. In the building process, the AFL compiler is

```
# Building Target Program
AFL_USE_ASAN=1 make CC=afl-clang-fast


# Building Fuzz Target
AFL_USE_ASAN=1 afl-clang-fast target.c -I../ \\
    ../libtoml.a -o target
```

LISTING 3.4: Build commands of AFL for tomlc99

```
#Running AFL
afl-fuzz -i in -o out -- ./target
```

LISTING 3.5: Run commands of AFL for tomlc99

used, the include path is extended to the path to the build folder, and the library to link against is specified. In both compilations, the variable AFL_USE_ASAN is set to 1 to use the address sanitizer.

Step 5 & 6: Run the fuzzer and interpret the result Afterward, the fuzzer is run by specifying the input and output folder and the target. The command is shown in Listing 3.5.

The fuzzer finds a crash in less than 5 minutes. The crash can further be analyzed, and it can be traced back to line 2014 in toml.c as the root cause.

libFuzzer and Suricata

In this example, libFuzzer is used to fuzz the open-source software Suricata. This example was also depicted in the previously published paper "A Qualitative Usability Evaluation of the Clang Static Analyzer and libFuzzer with CS Students and CTF Players," presented at the seventeenth Symposium On Usable Privacy and Security (SOUPS) in 2021, together with my co-authors Mischa Meier and Matthew Smith and was created with the support of Sirko Höer. The Fuzzer libFuzzer is used to fuzz the target program Suricata. This is a much more complex example in comparison to the previous one.

Step 1: Find a suitable function to fuzz Usually, parts of the code where the user input is used or parsed are good starting points. In the case of Suricata, I chose the AppLayerParserParse function. On the one hand, the function parses incoming packages and therefore opens up the possibility of covering larger parts of this specific part of the code. On the other hand, the function has a rather complex-looking structure with nesting and memory access which is prone to errors.

Step 2: Write the fuzz target Within the fuzz target, the *LLVMFuzzerInitialize* function is used to call several setup functions, and also a *LVVMFuzzerTestOneInput* function is written where a local setup is done with an initialization of a data package followed by the actual call of the function to fuzz where the data package had to be used. Hints for doing this could already be found in the existing AFL fuzz targets within the project. The fuzz target for Suricata can be found in Listing 3.6.

Step 3: Build and instrument the target program Instrumentation, as well as address sanitizer, are mandatory to trigger the bug in a reasonable amount of time. Since Suricata is not producing a library with the common build commands, a way has to be found to build Suricata such that the linking process to the fuzz target would be convenient.

The Makefile and target program are adjusted so that the *make* command directly builds the fuzzer. Also, a possible solution for that is to build the target program as described above and link the fuzz target later on against Suricata by, e.g., archiving all object files created by the common build commands in a library. In my opinion, the second solution is far less complex since it only requires a flag for building the target program, a one-liner to archive the object files, renaming the main function in the library, and a flag for compiling the fuzz target. On the other hand, altering Makefiles created by Automake can be very troublesome. The build commands for the target program and fuzz target can be found in Listing 3.7.

Step 4: Compile the fuzz target The command for compiling the fuzz target is depicted in Listing 3.8. the fuzzer flag, *-fsanitize=fuzzer*, is used to link the fuzz target agains libFuzzer. Also, the compiler is told where to find the library of the target program and the needed header files.

Step 5 & 6: Run the fuzzer and interpret the result The fuzzer is run by simply executing the executable build in Step 4.

The bug can be triggered depending on the corpus and chance within six to 24 hours. The output of libFuzzer when the bug is triggered is shown in Listing 3.9.

```c
#include <arpa/inet.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>

#include "suricata-common.h"
#include "app-layer-parser.h"
#include "flow-util.h"
#include "app-layer.h"
#include "app-layer-ssl.h"
#include "app-layer-dns-tcp.h"

int LLVMFuzzerInitialize(int *argc, char ***argv) {
  time_t t;
  MpmTableSetup();
  SpmTableSetup();
  AppLayerProtoDetectSetup();
  AppLayerParserSetup();
  AppLayerParserRegisterProtocolParsers();
   srand((unsigned) time(&t));
  return 0;
}

int GetDirection(){
    return rand() % 2;
}

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  if (size <= 63) return 0;

  // Data setup
  AppProto alproto = ALPROTO_TLS;

  size_t input_len = size;
  uint8_t *input = malloc(input_len);
  memcpy(input, data, input_len);

  Flow *f = NULL;
  TcpSession ssn;
  AppLayerParserThreadCtx *alp_tctx = AppLayerParserThreadCtxAlloc();

  memset(&ssn, 0, sizeof(ssn));

  f = calloc(1, sizeof(Flow));
  if (f == NULL) goto out;
  FLOW_INITIALIZE(f);

  f->flags |= FLOW\_IPV6;
  f->src.addr_data32[0] = 0x01020304;
  f->dst.addr_data32[0] = 0x05060708;
  f->sp = rand() % 65535;
  f->dp = rand() % 65535;
  f->protoctx = &ssn;
  f->proto = IPPROTO_TCP;
  f->protomap = FlowGetProtoMapping(f->proto);
  f->alproto = alproto;


  int start = 1;
  int flip = 0;

  uint8_t flags = STREAM\_TOSERVER | STREAM_START;
  SSLState *app_state = f->alstate;
  int r = AppLayerParserParse(NULL, alp_tctx, f, ALPROTO_TLS, STREAM_TOCLIENT,
                              input, input_len);
  if(r != 0){
    goto out;
  }



out:
  if (alp_tctx) AppLayerParserThreadCtxFree(alp_tctx);
  if (f) FlowFree(f);
  if (input) free(input);

  return 0;
}
```

LISTING 3.6: Suricata Fuzz Target

```
# Building Target Program
./configure \
    CFLAGS = "-O0 -g -fPIC \
    -fsanitize=fuzzer-no-link,address,undefined,signed-integer-overflow,bool,pointer-
        overflow \
    -fsanitize-coverage=trace-cmp,indirect-calls,inline-8bit-counters,pc-table,trace-
        div,trace-gep \
    -fprofile-instr-generate -fcoverage-mapping" \
    LDFLAGS = "-fprofile-instr-generate -fcoverage-mapping \
    -fsanitize=fuzzer-no-link,address,undefined,signed-integer-overflow,bool,pointer-
        overflow \
    -fsanitize-coverage=trace-cmp,indirect-calls,inline-8bit-counters,pc-table,trace-
        div,trace-gep \
    -fprofile-instr-generate -fcoverage-mapping"

make -j$(nproc)


# Patch Main
sed -i -e 's/main/mmmm/g' suricata.o


# Create Library
ar rv suricata_fuzz.a *.o
```

LISTING 3.7: Build commands for TPr of Suricata

```

# Building Fuzz Target
clang++ -O1 fuzz_app_ssl.c -I../src -I../libhtp \
    ../src/suricata_fuzz.a \
    -lmagic -lcap-ng -lpcap -lpthread -lnet -lyaml -lpcre -lz -llzma -llz4 \
    -fsanitize=fuzzer,address,undefined,signed-integer-overflow,bool,pointer-overflow \
    -o fuzzer
```

LISTING 3.8: Build commands for FT to fuzz Suricata

```
==79836==ERROR: AddressSanitizer: heap-buffer-overflow on address \
    0x62a00000dbff at pc 0x55d87348d19e bp 0x7ffed8d60730 sp 0x7ffed8d5fee0
WRITE of size 772 at 0x62a00000dbff thread T0
    #0 0x55d87348d19d in __asan_memcpy (suricata-5.0.0/fuzzing/app_ssl_bug/fuzz_app_ssl_bug+0x134e19d)
    #1 0x55d873a5e719 in SSLv3ParseHandshakeType suricata-5.0.0/src/app-layer-ssl.c:1476:13
    #2 0x55d873a52131 in SSLv3ParseHandshakeProtocol suricata-5.0.0/src/app-layer-ssl.c:1624:14
    #3 0x55d873a40f18 in SSLv3Decode suricata-5.0.0/src/app-layer-ssl.c:2297:22
    #4 0x55d873a2f3c1 in SSLDecode suricata-5.0.0/src/app-layer-ssl.c:2464:30
    #5 0x55d873a22051 in SSLParseServerRecord suricata-5.0.0/src/app-layer-ssl.c:2558:12
    #6 0x55d8739911c8 in AppLayerParserParse suricata-5.0.0/src/app-layer-parser.c:1239:13
    #7 0x55d8734c99a1 in LLVMFuzzerTestOneInput suricata-5.0.0/fuzzing/fuzz_app_ssl_bug.c:87:11
    #8 0x55d8733ca98f in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned \
        long) (suricata-5.0.0/fuzzing/app_ssl_bug/fuzz_app_ssl_bug+0x128b98f)
    #9 0x55d8733b14b7 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*, unsigned long) \
        (suricata-5.0.0/fuzzing/app_ssl_bug/fuzz_app_ssl_bug+0x12724b7)
    #10 0x55d8733b6588 in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, \
        unsigned long)) (suricata-5.0.0/fuzzing/app_ssl_bug/fuzz_app_ssl_bug+0x1277588)
    #11 0x55d8733a5e93 in main (suricata-5.0.0/fuzzing/app_ssl_bug/fuzz_app_ssl_bug+0x1266e93)
    #12 0x7f342825a151 in __libc_start_main (/usr/lib/libc.so.6+0x28151)
    #13 0x55d8733a5eed in _start (suricata-5.0.0/fuzzing/app_ssl_bug/fuzz_app_ssl_bug+0x1266eed)

Address 0x62a00000dbff is a wild pointer.
SUMMARY: AddressSanitizer: heap-buffer-overflow \
    (suricata-5.0.0/fuzzing/app_ssl_bug/fuzz_app_ssl_bug+0x134e19d) in __asan_memcpy
Shadow bytes around the buggy address:
  0x0c547fff9b20: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c547fff9b30: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c547fff9b40: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c547fff9b50: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c547fff9b60: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c547fff9b70: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa[fa]
  0x0c547fff9b80: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c547fff9b90: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c547fff9ba0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c547fff9bb0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c547fff9bc0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==79836==ABORTING
```

LISTING 3.9: Output Libfuzzer Suricata Bug

Chapter 4

Usability Evaluation of a Static Analysis Tool and libFuzzer

Disclaimer: *The contents of this chapter were previously published as part of the paper "A Qualitative Usability Evaluation of the Clang Static Analyzer and libFuzzer with CS Students and CTF Players," presented at the seventeenth Symposium On Usable Privacy and Security (SOUPS) in 2021, together with my co-authors Mischa Meier and Matthew Smith. As this work was conducted with my co-authors as a team, this chapter will use the academic "we" to mirror this fact. I designed the study with advice from Matthew Smith. I set up and performed the study as well as designed and conducted all interviews. Mischa Meier supported me in the coding and the discussion of the interviews.*

Finally, before compiling the paper for publication, Matthew Smith and I jointly discussed the study's implications.

4.1 Motivation

The number of critical security vulnerabilities is rising, with the same type of programming mistakes being made over and over again. Testing software for bugs and vulnerabilities is one crucial aspect of helping developers write secure code and countering this development.

The two prevalent approaches for application security testing are static analysis and dynamic analysis.

Static analysis has seen widespread adoption across the industry, dominating the leaders' portfolio of the April 2020 Gartner magic quadrant for application security testing [73]. Dynamic analysis, and in particular fuzzing, has received much attention in academia in recent years, as can be seen by this selection of fuzzing papers published in 2020 alone: [115, 206, 142, 207, 149, 68, 162, 148, 96, 31, 111, 71, 209, 151, 203, 122, 98, 154, 187, 192, 123, 20, 197, 72, 143, 145, 22, 166, 14, 105, 189, 32, 57, 7, 90, 198, 153]. Moreover, large software companies such as Google, Microsoft, Cisco, and others use fuzzing very successfully. For instance, using fuzzing, Google found over 20,000 bugs in Chrome alone [26]. Despite these impressive results, fuzzing has not yet found the same adoption in industry that static analysis has.

In this paper, we examine the usability of the static analyzer Clang Static Analyzer and the fuzzer libFuzzer to get first insights into the question of whether usability issues might be hindering the adoption of fuzzing. For our study, we evaluated several fuzzers and static analyzers. We selected the Clang Static Analyzer because it performed very well in the comparison of Arusoaie et al. [6] and libFuzzer because it is a popular example of a dynamic code analysis tool in academia [155, 96, 17]. However, we would like to stress that neither the Clang Static Analyzer nor libFuzzer are necessarily representative examples of static and dynamic analysis tools. Moreover, since the tools are good at finding bugs of different types, our evaluation should not be seen as a like-for-like comparison but as gathering first insights into usability strengths and weaknesses of two different tools.

We performed a qualitative mixed factorial design study with 32 CS master students and six competitive Capture the Flag (CTF) competitors to evaluate the usability of the Clang Static Analyzer and libFuzzer with an easy and a hard task. We designed an easy and a hard task to get a broader view of the tools. The easy task was designed to see if participants could get the tool running in principle, while the hard task was designed to reflect a more realistic challenge as would be faced in a real project. The two tools were studied using a within-subjects design to also gather comparative insights into the two tools. The difficulty of the tasks was tested between subjects with the CS students. The CTF participants only got the hard task. Participants had ten hours over a period of ten days per task to work on the solution.

Our results indicate that the Clang Static Analyzer is easy to use in principle, but it did not scale well to the hard task. Only one CTF participant was able to find the bug due to a large amount of false positive warnings. With libFuzzer, the usability hurdles were much higher, and many CS participants did not manage to solve even the easy task. Even the CTF players did not manage to find the bug in the time allotted, although they were able to use libFuzzer in principle. While the majority of participants only failed in the last step of the Clang Static Analyzer, we found usability problems in every step needed to use libFuzzer, which we will discuss throughout the paper.

4.2 Methodology

We wanted to gain insights into the usability issues of the Clang Static Analyzer and libFuzzer. In the following, we will discuss the design and methodology of the two studies we conducted to do this.

4.2.1 Tool Selection

We decided to pick one tool per category instead of a spread since it was expected that we would not be able to recruit enough participants to compare multiple tools.

Static Analysis We evaluated the popular commercial static analysis tools Fortify [125], Coverity [44], CodeSonar [41] and checkmarx [29]. Unfortunately, they all forbid publishing evaluations in their terms of use [126, 45, 42, 30]. We based our selection of the open-source static analysis tool on the evaluation of Arusoaie et al. [6]. Also, we took the new analyzer ikos [19] into consideration as it was not part of their evaluation. We concluded that ikos was not yet stable and production-ready and removed it from the list of possible candidates. Based on this, we selected the Clang Static Analyzer, which was the analyzer with the highest productivity rate, a combination of detection rate and false-positive rate, and the highest win rate combining all subcategories within their analysis. We selected the Clang Static Analyzer in version 8.0 as it was the latest version at the time we conducted the first study.

Dynamic Analysis When designing the study, there were no popular commercial fuzzers for C/C++ code available, so we only evaluated the open-source fuzzers: AFL [4], AFL++ [5], libFuzzer, honggfuzz [88] and radamsa [159]. Our literature review showed that AFL/AFL++ and its forks, as well as libFuzzer, are the most common fuzzers in use [120, 9, 33, 187, 14, 67, 96, 17]. Both AFL and libFuzzer were viable choices. While both Fuzzers can fulfill the same tasks, we think that both have strengths and weaknesses for specific situations. To fuzz with libFuzzer a specific function is picked as an entry point. In contrast, AFL primarily works on and with executables. The fuzzer itself is a standalone executable that takes the executable of the target program as input. This provides the user straightforward access to the fuzzing process, and it helps to get a first impression of the robustness of a target program. On the other hand, it only allows fuzzing a target program from the main entry point, leading to the case that the Fuzzer might not reach specific parts of the code. In the hard task, it is unrealistic that the code section containing the bug can be reached by AFL this way, while libFuzzer can be run directly on the function. For this reason, we choose libFuzzer over AFL.

4.2.2 Task Selection

To evaluate the usability of the tools, we needed programs containing vulnerabilities that participants should find. While we were also interested in comparing the usability of the Clang Static Analyzer and libFuzzer, it was not feasible to use the same vulnerabilities for both tools since the types of bugs these tools are good at finding vary too much. We were also interested in comparing how the tools performed at different levels of difficulty. We chose one easy task per tool to get a baseline. With that, we could uncover fundamental difficulties with the tool itself. Additionally, a hard task was chosen per tool to see how it performed in a more realistic setting.

Prerequisites

An appropriate task, i.e. a program to be analyzed, needs to contain a vulnerability that the respective analysis tool can find. This bug should be hard to find by other means than using the tool, particularly by using search engines, thus data-sets like the DARPA Cyber Grand Challenge [54] were not viable options for us. We also decided against using tools like Lava-M [61] since they generate a recognizable style of bugs that we knew were familiar to the CTF participants, and inserting bugs into existing programs opens up the risk that participants could use the DIFF tool to identify changes quickly. Ideally, we could use actual undiscovered bugs. To make the matter more complicated, it was also desirable that the difficulty of the two easy and two hard tasks would be similar.

Static Task

We started by running the Clang Static Analyzer on several trending GitHub projects at that time. A list can be found in the Appendix in Table A.1. While most of the projects had a high number of warnings, we could not find any true positives, despite investing a significant amount of effort into this. Since this proved fruitless, we contacted experts in static analysis from the Cyber Analysis and Defense and the Cyber Security research departments of the Fraunhofer FKIE to discuss program selection. They did not have any fixed but unpublished bugs, so we were unable to find an unpublished bug suitable for our study. Thus, we fell back on inserting vulnerabilities ourselves but attempting to mitigate the issues mentioned above. For this, we injected one bug in a local copy of the open-source project jq [101] for the easy task and two bugs into a local copy of the open-source project Tesseract [178] for the hard task. The injected bugs were never deployed anywhere outside the study and did not endanger anybody. We chose these projects based on the number of warnings since related work showed that the number of false positives is the main usability issue of static analyzers. Project jq only produced five warnings, and we checked all to confirm they were false positives. Tesseract produced 476 warnings, and we did not find any true positives. We chose to inject one bug, which the Clang Static Analyzer can find without any options activated in both programs. We also injected an additional bug in the hard task, which requires the tester to set the checker *alpha.security.ArrayBoundV2* manually to inspect array boundaries. To mitigate the risk of participants using DIFF to find the inserted bug, we chose older versions of the programs and removed all information concerning the version number.

Dynamic Task

Unlike with the Clang Static Analyzer, there was no simple way with libFuzzer to evaluate a set of GitHub projects similarly, so we contacted Code Intelligence, a company offering fuzzing as a service to get an overview of difficulty levels of different projects. Fortunately, they knew a couple of open-source projects with vulnerabilities that had already been reported and fixes submitted but not publicly announced yet. Hence, we selected two of these for our fuzzing tasks. For the easy task, we used yaml-cpp [202] since it is a comparatively small project and has only a handful of public interfaces. This circumstance makes it reasonably easy to get a good overview of the program in a moderate amount of time. Also, writing the fuzz target is relatively simple, and the bug is found in a couple of seconds, even without instrumentation. We knew of one bug in yaml-cpp.

For the hard task, we selected the Suricata [175] project. The fuzz target needed to trigger the bug is more complex than for the yaml-cpp project, and instrumentation, a fitting corpus, and time are needed. Based on the fuzzing expert's recommendations, we opted to give a starting hint to give participants an idea of where they should start looking since the code base was huge, and it would take more time than was available in the study to get an overview. We knew of two bugs in the location where we gave a hint. We fixed one of them since it was a very easy bug, and this was supposed to be the hard task. There were also two other bugs in a different code section. However, these were not relevant to our study. So for the purpose of this study, we had one bug in the location for which we gave the hint. In addition to our hint, the Suricata project contained two other sources participants might use to guide their fuzzing effort. The project contained unit tests that could be adapted into fuzz targets. The projects also contained some AFL fuzz targets. As far as we could tell, triggering the bugs with the AFL fuzz targets was impossible.

4.2.3 Study Design CS Study

Our study contains two independent variables, each with two levels: analyzer (Clang Static vs. libFuzzer) and difficulty (easy vs. hard). Based on our external experts' feedback and internal pre-studies, we decided to allot ten hours for each of the four study conditions. Since this study is highly skill-dependent, we opted for a within-subjects study design for the analyzer variable. To reduce the time needed per participant, we opted to study the difficulty level between-subjects, which then gave us a mixed factorial study design. So each participant either did the easy task with both the Clang Static Analyzer and libFuzzer or did the hard task with both analyzers. We randomly assigned the participants to the hard or easy tasks and randomized the order in which participants used the analyzers to counter learning and fatigue effects. Due to the length of the tasks and the fact that fuzzers need to run for a while to find bugs, we conducted the study online. Participants had ten days per condition and were instructed to work ten hours. If they thought they had found all bugs, they could report in early and would then be given the second task. Participants were asked to keep a diary while working on the task detailing what they spent time on and what problems they encountered. We supplied remote virtual machines with the tools pre-installed for participants to use. They were, however, also allowed to work on their machines if they preferred. After completing both tasks, participants took part in a 30-minute semi-structured interview.

Recruitment and Participants

Since our study required a time commitment of 20 hours, recruiting enough professional developers was not feasible for us at this stage of our research. Thus, we opted to use CS students from a lecture on usable security and privacy and CTF players to gain first insights but want to point out that professional developers would probably perform better in absolute numbers. However, fixing the usability problems discovered by the CS students is likely also to be beneficial to professional developers. However, we cannot make any claims to this extent. Additionally, CS students are also a legitimate user group for these tools, and consequently, fixing usability issues for them is also a desirable goal.

The lecture is part of a master of computer science curriculum and is not mandatory. The focus of the lecture is usability in the context of security. Consequently, all participants had a bachelor's degree in computer science, had some basic knowledge of how to evaluate the usability of security tools.

Since the tasks require C/C++ and Linux skills, we used a pre-questionnaire as a filter. We selected a self-reported skill level for Linux and C/C++ of four or higher on a scale of one to seven. We distributed the pre-questionnaire to about 110 students and selected 32 for the study who fulfilled the requirements. They were compensated with an 11% bonus for their end-of-term exam. Students not selected for this study had other opportunities to earn the same bonus. Table 4.1 shows the demographics of the CS student participants.

Gender	Male: 26	Female: 5	Other: 0	No Answer: 1
Age	min: 22, max: 34	mean: 26.03, median: 25	sd=2.95, NA=0	

TABLE 4.1: CS Participant Demographics

Only six out of the 32 participants reported that they had ever used a static code analyzer before. 17 participants reported that they were familiar with the term fuzzing. However, only four of them had used a fuzzer before. Three of the participants had found a bug with a fuzzer, and one had used the fuzzer libFuzzer before.

4.2.4 Study Design CTF Study

The second study conducted with CTF players was designed and run after the results of the first study with CS students had been evaluated. While the studies are very similar, we did make three changes which we will highlight here.

Firstly, based on the results of the CS study and the expected skill level of the CTF players, we dropped the easy tasks since we did not expect to learn much there and focused on the hard task, as it would be more beneficial also considering the small pool of CTF players.

Secondly, the Suricata project released an update fixing two of the three bugs we knew of, and information about them had been released. To prevent participants from quickly finding these two bugs via a web search, we gave our participants the updated version, which thus only contained one bug we knew of for them to find. Fortunately, this bug was the one in the code section for which we gave a hint so we could leave the task unchanged except for the update.

Finally, since exam bonus points were not an option, we offered monetary compensation instead. We initially offered a base compensation of 70 euro, with an additional 70 euro offered for finding bugs. We thought due to the competitive nature of CTF players, they would respond well to the incentive. However, we were not able to gain enough interest in our study. After talking to some potential participants, we switched to a flat compensation of 140 euro independent of success.

Recruitment and Participants

We recruited participants from a local Capture-the-Flag (CTF) team via announcements in the weekly team meetings and email. This pool contains roughly 80 people, of which 16 filled out the pre-screening survey. We removed participants who did not have at least one year of CTF experience and had taken part in at least one online and one in-person CTF challenge since we wanted to have a highly skilled group for comparison with the CS students. This left us with eight participants who took part in the main study. During the study, it turned out that two participants had misunderstood the question about in-person CTF events. They actually had not taken part in any and thus are not included in this report.

The demographics of the CTF group are shown in Table 4.2.

Gender	Male: 8	Female: 0	Other: 0
Age	min: 19, max: 32	mean: 23.25, median: 22	sd=4.2, NA=0
Level of education	Bachelor: 2	apprenticeship: 2	high-school diploma: 4
CTF events online	min: 1 , max: 20+	mean: 11.4, median: 12.5	sd=8.4, NA=0
CTF events offline*	min: 1 , max: 7	mean: 4.2 , median:4	sd=2.6, NA=0

TABLE 4.2: CTF Participant Demographics

Compared to the CS group, the CTF group is younger, more male and the education level is slightly lower.

Similar to the CS group, only two of the six participants had used a static code analysis tool.

However, all participants reported that they were familiar with the term fuzzing. Five of them had already used a fuzzer before, and three of those five participants had also used libFuzzer and half of all participants reported that they had already found at least one bug with a fuzzer.

This indicates that the CS and CTF group were on a similar level w.r.t. static code analysis tools, but the CTF group had more experience with fuzzing.

4.2.5 Scoring Results

We evaluated the analyzers based on the success or failure of the participants to get the tool up and running as well as finding the bug. These are separate since it is possible to correctly use the tool but still fail to find the bugs. To make our assessment, we analyzed the submissions of the participants (code and bug reports) as well the content of the diary and the exit interview. The fulfillment is a binary variable with the possible values of success and no success.

In the static analysis case, a participant successfully fulfilled a static task if the participant used the Clang Static Analyzer correctly and found at least one of the bugs we inserted.[1]

A participant successfully fulfilled a dynamic task if the participant triggered the bug present in the code by using libFuzzer and recognized it as a bug.

4.3 Limitations

Our studies have the following limitations:

Task Selection. The most considerable limitation concerns the task selection. While we did our best to find fair easy and hard tasks for both tools and consulted external experts, we cannot guarantee that the two easy and hard tasks are exactly the same level of difficulty. While the identified problems likely remain for other tasks, the difference between the two approaches could vary for different tasks.

Participants. We sampled participants from a master course in usable security and a CTF team. Thus this sample is not representative of the wider world. Nonetheless, fixing the issues we found is most likely a good idea, even if more experienced developers might have learned to overcome them.

[1] Another true positive bug would also have counted, but this did not occur.

Tools Selection. We tested two specific tools: Clang Static Analyzer and libFuzzer. Other tools might perform differently.

Time. Participants only had ten hours per task. While our internal testing suggested that this would be sufficient, some CTF participants would likely have found the bug with libFuzzer, since they were making progress until the end. The time limit did not seem to affect the CS participants or the static tasks.

Unknown Code. Our evaluation only looks at participants analyzing code that they did not write themselves. Further studies with code known to the participants are needed to make claims about this scenario.

Incentives. When comparing the CS and CTF groups, the different incentives must be taken into account.

Bugs. During the second study with the CTF participants, information about the bugs in Suricata was published in a blog. One of our participants found this blog and informed us about it. We contacted the author of the blog, and they kindly agreed to take it offline until the end of the study. The participant who informed us about the blog had already finished the task. We asked all other participants whether they had come across the blog or other information online. One additional participant had found some information in an online presentation; however, this did not help them complete the task.

4.4 Ethics

Our studies were reviewed and approved by the Research Ethics Board of our university.

Our studies also complied with the General Data Protection Regulations. Since we were working with live vulnerabilities, responsible disclosure guidelines were followed. The developers of both programs were already aware of the Bugs, and all participants agreed to comply with responsible disclosure in case they found bugs.

4.5 CS Study Results

We label participants based on their group (CS or CTF), the order of assignment to the conditions ((FS: fuzzing then static, SF: static then fuzzing), and the difficulty of their tasks(E: easy, H: hard). For the analyses, we used the pre-questionnaire, the reports submitted by the participants, the diaries, and the semi-structured post-interview. The questions of the interview and the pre-questionnaire can be found in the Appendix in Section A.1.1 and A.5.1. Except for CS16-FSE, every participant consented to the interview being recorded and transcribed. For the interview of CS16-FSE, handwritten notes were taken. The interviews were transcribed and anonymized.

To analyze the interviews and diaries, we used inductive coding [179] with two researchers. The two researchers started by coding the same four randomly chosen interviews independently and in parallel. They compared, analyzed, and discussed the two resulting coding sets. It turned out that due to the open approach, the code sets of both researchers were substantially different. Through a discussion of the codes, a common coding set was agreed upon. The four interviews were then recorded and discussed again. This procedure was repeated in steps of three interviews. The diaries of the participants were coded with the resulting coding set from the coding of the interviews. During the coding of the diaries, the coding set was again supplemented by codes that emerged from the data. All quotes from the participants were translated from German into English by the authors.

4.5.1 Drop-outs

Of the 32 CS student participants, only 18 started the second task, and only ten finished both tasks and were interviewed. CS18-FSH finished both tasks and took part in the interview, but we decided to remove them from our analysis because it became clear that they had not put any real effort into either task. This leaves us with nine participants that finished both tasks and were interviewed. The drop-out rates were much higher than we expected. We have conducted many usability studies with CS students, and it is normal that some drop-out, but this drop-out rate is noteworthy. While we did not conduct formal interviews with the drop-outs, we spoke to some of them. They told us that the tasks were too hard and that they did not know how to solve them and thus dropped out.

The second column of Table 4.3 shows the drop-out rates, and, as can be seen, only a quarter of the participants dropped out of the easy static task, while half dropped out of the hard static task. With the fuzzing tasks, half dropped out both in the easy and hard tasks. This is a first indication that there are usability issues with both approaches. While this explanation seems plausible, based on the rest of the data we could gather, it is also possible that the drop-out rate could be an artifact of our study design. Further studies with different designs are needed to confirm this.

Since we were also interested in a qualitative within-subjects comparison of the Clang Static Analyzer and libFuzzer, most of our analysis focuses on the nine CS participants who completed both tasks and who were interviewed. Table A.7 shows an overview of the participants' positive and negative comments and their preference for the two tools. In the following, we look at the results in more detail.

4.5.2 Static Task

The results of the static analysis tasks can be seen in Table 4.3. Table A.2 in the appendix breaks the results down into those who were assigned the conditions as their first or second task. As can be seen, the easy task was indeed relatively easy, with only three participants aborting the task. Moreover, in eight out of nine submissions in the easy tasks, the bug was correctly identified. In contrast to that, half the participants dropped out of the hard task. Of those who submitted a report for the hard task, none had found either of the two bugs. In the following, we will group our insights by the different steps needed to complete the task.

Step1: Build Target Program with Clang Static Analyzer All participants, except for CS23-SFE, started by reading the documentation of the Clang Static Analyzer. CS23-SFE had used the analyzer before and knew the necessary steps. None of the participants reported that they used any other source of information besides the documentation of the Clang Static Analyzer and the target program(TPr).

Not many participants had problems with this step, except for two participants, CS31-SFE and CS24-FSE. Both had used the *configure* and *make* commands on the project to check if everything worked as intended. This interfered with the Clang Static Analyzer because the target program was already built. Therefore the analyzer could not build the target program again and consequently could not find any bugs. CS24-FSE solved this problem on their own. CS31-SFE submitted a report stating that no bugs were found. To gather more information, we let CS31-SFE know that something went wrong and gave a hint. CS31-SFE still counts as a fail in the overall statistics, but with the hint were able to complete this step, and their results are considered in the following steps.

Step 2: View the Output Five out of the nine participants who submitted a report had trouble viewing the output of the Clang Static Analyzer. However, this problem only arose because the participants were working on the remote machines offered by us. Except for CS31-SFE, all participants solved the issue by downloading the output to their local machines. Since this problem stemmed from our study setup, we do not see this as a usability issue of the tool.

Step 3: Analyze Reports The presentation of the output of the Clang Static Analyzer was rated very positively by the participants. However, as expected, all participants in the hard task and some in the easy task stated that the massive number of warnings was a substantial problem. In particular, the high number of duplicate bug reports was viewed negatively. This is in line with previous work looking at static analyzers. What is noteworthy though is that this problem has been well known for over a decade but is still an issue with current tools.

4.5.3 Dynamic Task

The results of the dynamic analysis tasks in Table 4.3 show that both tasks were hard to solve for our CS participants. For a more detailed overview showing in which order the tasks were assigned, please refer to Table A.5 in the Appendix.

Only two CS participants were able to solve the easy task. CS6-FSE dropped out in the following static task, but their diary showed that they straightforwardly solved the task mentioning no problems. The other participant was CS23-SFE, who had stated that they already had experience with fuzzing and libFuzzer in particular. Another participant, CS5-SFE, wrote the correct fuzz target and ran the fuzzer triggering the bug but was convinced that the fuzzing report did not describe a bug.

None of the participants was able to solve the hard task. The drop-out rates for both fuzzing tasks were roughly half, just like for the hard static task.

combined	started	drop-out	submitted	success
Static-easy	12	3	9	8
Static-hard	10	5	5	0
Fuzzing-easy	16	8	8	2
Fuzzing-hard	10	6	4	0

TABLE 4.3: CS static analysis and fuzzing overview

Unlike with the Clang Static Analyzer, which almost all participants used correctly, we found many problems with the usage of libFuzzer. Table 4.4 gives an overview of where participants had problems. The columns of Table 4.4 depict the six steps of the fuzzing process. The first step of finding a suitable function to fuzz contains two values. The first value is the number of functions a participant tried to fuzz. The second value indicates if the participant found a function that triggers one of the bugs known to us. The step of building and instrumenting the target program also contains two values. The first value indicates whether the target program was built, the second if the target program was instrumented. The other columns indicate: how many fuzz targets were created, whether they could build the fuzz targets, whether they ran the fuzzer, triggered the bug, interpreted the output correctly (either as false or true positives), used a corpus and used toy examples to try out fuzzing before trying it on the main project.

The first nine participants in the table are those who completed all tasks and the interview. The next participant in blue is the low-effort participant. The participants below in gray completed the fuzzing task but then dropped out. Since we conducted the study online, and participants were allowed to use their own computers, we could not always reconstruct every step. When we were uncertain about whether a participant successfully took a step or not, we marked this with a circle.

Participant	Condition	Found Func.	Wrote FT	Build & Inst. TPr	Build FT	Ran Fuzzer	Bug Trig.	Interp. Output	Corpus	Toy
CS16-FSE	easy	✗ / ✗								✗
CS31-FSE	easy	✗ / ✗								✓
CS15-SFE	easy	2 / ✓	2	✗ / ✗ (FT in TPr)	✗					✗
CS24-FSE	easy	1 / ✓	1	✓ / ✗	✗					✓
CS5-SFE	easy	1 / ✓	1	✓ / ✗	✓	✓	✓	✗	✓	✗
CS23-SFE	easy	1 / ✓	1	✓ / ✓	✓	✓	✓	✓	✓	✗
CS4-FSH	hard	✗ / ✗								✓
CS3-SFH	hard	❍ / ❍	❍	✗ / ✗	✗					✓
CS8-FSH	hard	1 / ✗	❍	✗ / ✗						✓
CS18-FSH	hard	✗ / ✗								✗
CS28-FSE	easy	✗ / ✗								
CS6-FSE	easy	2 / ✓	2	✓ / ✗	✓	✓	✓	✓	✗	✗
CS17-SFH	hard	✗ / ✗								✗
CS30-FSH	hard	✗ / ✗		✗ / ❍						
CS26-FSH	hard	❍ / ❍	❍	❍ / ❍						✓

TABLE 4.4: CS dynamic analysis deeper statistics: ✓ denotes success in this phase, ✗ failure and ❍ undecidable

For our qualitative analysis, we again focus on the participants that finished both tasks and were interviewed. In the following, we will group our insights by the separate steps needed to complete the task.

Familiarization with the Process All participants started with getting an overview of libFuzzer as well as the target program. Unlike the Clang Static Analyzer, where participants only used the official documentation, many participants searched for additional information about libFuzzer on the web. This highlights deficits in the official documentation as emphasized by CS5-SFE:

> So if you visit the [libFuzzer] page, it is not really obvious what you need to do.

and by CS15-SFE:

> I have not used a fuzzer and I would have wished for a guideline. Such as: Step one, do this, step two, do this... getting started was really hard.

Moreover, participant CS15-SFE stated that the documentation negatively impacted them:

> Even after reading through the paragraphs several times, I'm not sure where to start. Instantly start to lose interest.

Step 1: Find a Suitable Function to Fuzz In this step, participants had to identify functions to fuzz. In the easy task, all participants who identified any functions to fuzz also identified the one that could trigger the bug. Three participants, CS16-FSE, CS31-SFE, and CS4-FSH did not find any functions they thought they could fuzz. CS4-FSH summarized the problems with:

> I looked at the source code of Suricata and was completely overwhelmed. [...] And in the end I did not find any approach how I could fuzz this with a fuzzer.

CS31-SFE commented on that:

> I had problems finding the right point to start fuzzing. The website was not much of a help: [...]. Should I try to look at it from an external view and try to feed information from the outside or should I do it internally [...]? I was missing many examples. It would have been good to not only see somebody fuzzing an easy function [...].

Step 2: Write a Fuzz Target All participants in the easy task who found the function to fuzz also successfully wrote the correct fuzz target. None of the CS participants managed to write a correct fuzz target in the hard task.

Two participants, CS15-SFE and CS3-SFH, tried to write the fuzz target in an existing file of the target program. CS3-SFH changed their mind after having problems with the compilation and used an external fuzz target. For CS15-SFE, this resulted in a more complicated situation. They had to remove the corresponding main function of the target program to use libFuzzer since libFuzzer is shipped with a main function, which interferes with other main functions. More importantly, they also had to modify the make file in order to compile the altered target program. This seemed to have been motivated by the code snipped in the official documentation that could give the impression that the fuzz target is part of the target program. CS3-SFH also stated to this topic:

> I tried to write a simple fuzzer target for a function in app-layer-parser. I started simple and did not manipulate the inputs. I directly wrote it into the app-layer-parser.c file like in the examples given...

CS3-SFH did not include the fuzz target in the report, so we could not confirm this.

While this is a legitimate way to run libFuzzer, in our view writing the fuzz target in a separate file is a cleaner and more straightforward approach.

Step 3: Compile and Instrument Target Program The actions taken in the step of compiling and instrumenting the target program were wild. In the case of the easy task, none of the CS participants, except CS23-SFE, seemed to be aware that instrumentation exists or had any idea why instrumentation is useful. CS23-SFE was the only participant who actively dealt with instrumentation and was aware of the implications of the "fuzzer-no-link"-flag and was the only one ever to use it.

All participants of the hard task had the problem that Suricata builds as an executable, and libFuzzer can not directly fuzz executables. CS8-FSH tried to find a solution by exporting a function from the Suricata elf binary into a shared object and then load and run it within a fuzz target. However, they were not able to do so.

Step 4: Build Fuzz Target The five remaining participants reported severe problems in the building and linking step. CS24-FSE stated:

> I believe that the library itself wasn't the problem, but the stupefying linking and compiling was.

The problems with building and linking could have a variety of reasons. Two participants stated that they lacked knowledge concerning the make system (CS24-FSE, CS15-SFE) or even compiling C/C++ code in general (CS5-SFE). Other participants had problems linking libraries and were randomly trying out compiler and linker flags to get the fuzz target to compile. For yaml-cpp, some participants also tried to use *make install* on the target program to increase the chance of hitting the right combination of compiler and linker flags. Overall, we observed a lack of understanding concerning the interaction between fuzz target, target program, and compilation process.

Step 5: Run and Observe the Fuzzer In the easy task CS5-SFE, CS6-FSE, and CS23-SFE were able to build the fuzz target and run libFuzzer. Moreover, they all triggered the bug because, in the easy task, the bug was triggered within seconds.

Step 6: Interpret Output Of the three participants who triggered the bug, CS5-SFE incorrectly classified the output as a false positive. CS5-SFE saw the out-of-memory error and the malformed input the fuzzer had generated but thought this was a mistake by libFuzzer instead of a bug in the program.

Even though CS23-SFE was by far the best participant in solving the easy fuzzing task in less than two hours, they did not find the output of libFuzzer very helpful, stating:

> I would be helpful if the output did not just contain the input which led to the bug, but also information about the crash.

Toy Examples and Documentation Six of the nine participants experimented with the toy examples from the documentation to get to know libFuzzer. However, as described above, this led some astray.

Moreover, almost all participants stated that the documentation was lacking in many different ways. The two most prominent complaints were the lack of precise steps and the request for more complex and well-explained examples.

4.6 CTF Study Results

The interviews and diaries of the CTF group were coded based on the same principles we used for the CS group. The questions of the interview and the pre-questionnaire can be found in the Appendix in Section A.1.2 and A.5.2.

An overview of the CTF-group's success can be seen in Table 4.5. Unlike in the CS group, we had no drop-outs in the CTF group. There are two potential explanations for this. Based on our interviews, the CTF participants were not as frustrated with the tools as the CS participants or had a higher frustration threshold and a willingness to work with complicated and puzzling systems. However, it could also be that the 140 euro incentive was more motivating than the 11% exam bonus or a combination of these factors. As in the previous section, we will structure our results around the steps needed to operate tools.

combined	started	dropout	submitted	success
Static-hard	6	0	6	1
Fuzzing-hard	6	0	6	0

TABLE 4.5: CTF: overall results

Static Analysis

Steps 1 & 2 The participants had no problems getting to the point where they had to inspect the reports given by the Clang Static Analyzer. Some participants reported issues viewing the results, like in the CS study, but could quickly solve them.

Step 3: Analyze Reports Overall, participants were satisfied with the usability of the tool as with the presentation of the output but had the same problem with the high number of false positives as the CS group. CTF7-SF stated:

> More than once I wondered whether it's me or the analyzer who doesn't understand the code.

Only one participant (CTF2-FS) was able to find one of the bugs.

Notably, four out of six participants reported that they heavily prioritized reports in the category *memory errors*. Some specifically mentioned that they neglected reports in other categories, such as *Logic errors*, which was the category where the Bug was. Their reasoning was that these kinds of bugs potentially have low exploitability. In the interviews, some of the CTF participants stated that they did not consider availability/denial of service an issue in this context. This could be an artifact of the fact that in CTF games denial-of-service attacks are often forbidden. CTF7-SF stated:

> Going through the "Memory error" bugs - If there are any vulnerabilities I expected to find them here, so I took some time for them.

All in all, participants showed strong tendencies to focus on bug types, ignoring much of the output produced by the Clang Static Analyzer. CTF3-SF summarized it as follows:

> I filtered for use-after-free and double-free/delete, which seemed most likely to have immediate security impacts. While there were 72 bugs shown in total, most of them were duplicates. I decided to only look at one bug per bug group/function combination, which eliminates mostly very similar code paths... For each combination, I chose the shortest path length to have a minimum-complexity example of a triggering code path.

This filtering caused the participants to miss our bug, which was in the category *Dereference of undefined pointer value*.

Dynamic Analysis

Despite being more experienced and security savvy, our CTF participants also had trouble with libFuzzer. Table 4.6 gives an overview of where participants had problems.

Participant	Found Func.	Wrote FT	Build & Inst. TPr	Build FT	Ran Fuzzer	Bug Trig.	Interp. Output	Corpus	Toy
CTF5-SF	1 / ✓	1	✗ / ✗	✗					✗
CTF4-FS	1 / ✓	1	✓ / ✓ (FT in TPr)	❍	❍	✗	✓		✗
CTF2-FS	1 / ✓	1	✓ / ✓ (FT in TPr)	✓	✓	✗	✓	✓	✗
CTF6-FS	1 / ✓	10	✓ / ✓ (FT in TPr)	✓	✓	✗	❍	✗	✗
CTF7-SF	1 / ✓	11	✓ / ✓	✓	✓	✗	✓	✓	✗
CTF3-SF	1 / ✓	3+	✓ / ✓ (FT in TPr)	✓	✓	✗	✓	✓	✗

TABLE 4.6: CTF dynamic analysis deeper statistics: ✓ denotes success in this phase, ✗ failure and ❍ undecidable

Step 1: Find a Suitable Function to Fuzz Unlike the CS participants, all CTF participants were able to identify the correct function to fuzz.

Step 2: Write Fuzz Target The writing of the fuzz target split the CTF group in two. Participants CTF4-FS and CTF5-SF used the unit tests as the basis for their fuzz targets. Participants CTF2-FS, CTF3-SF, CTF6-FS, and CTF7-SF based their fuzz target on the AFL targets contained in the project. In general, all participants agreed that creating the fuzz target was a complicated and time-consuming task.

Step 3 & 4: Compilation and Instrumentation Five out of six participants were successful in compiling and instrumenting all necessary parts, only CTF5-SF did not successfully manage this step. CTF5-SF had criticism for the documentation and some suggestions on how the usability of these steps could be improved.

> Although everything was described [in the example] instructions were missing how to approach fuzzing a real-world project, how to integrate it into an existing boot-system. Maybe one could have made something generic to integrate it into Cmake or Auto-build.

Four of the five participants who created a fuzz target wrote the fuzz target directly into the target program. Unlike the CS participants, they were able to make the necessary modification to make this work. We found this interesting since it does not seem to be the intuitive way for us.

Step 5: Running and Observing the Fuzzer The four remaining participants, CTF2-FS, CTF3-SF, CTF6-FS, and CTF7-SF, created multiple fuzz targets and observed the fuzzing process.

All participants focused on using the executions per second as well as the code coverage as the indicators on whether the fuzzing process was going well or not. Concerning the code coverage, some participants mentioned that it could sometimes be hard to interpret the relative magnitude of the given value correctly. CTF6-FS summarized it as follows:

> Of course, this depends on the complexity [of the TPr], but when I have such an HTTP fuzzer, and I know it is implemented in C, and I only have twenty branches or so which have been covered, then I know: This can't be. This absolutely can't be! You can't implement an HTTP fuzzer with so few branches or so few basic blocks. And if it also isn't making progress, then, you need to find out what is the matter.

The problem of knowing whether libFuzzer covered the necessary parts of the code was a frequently reoccurring statement. Only CTF2-FS used the visualizer of LLVM to get a better understanding of the situation.

Step 6: Interpret the Output CTF4-FS wrote a fuzz target and was also able to build it. However, the fuzz target quality was relatively low, so that the fuzz target crashed directly due to problems during initialization when executed. The participant was aware of the problem but could not fix it. CTF4-FS stated:

> And when I wanted to fuzz the correct filter, I always failed because something was uninitialized and this was why it always crashed. So it always fuzzed but crashed in each attempt.

Unsurprisingly, CTF4-FS believed that fuzz target creation was a big problem. They reasoned that this might partially be because they did not know the code.

> It was probably because I didn't know the software at all and then I couldn't proceed as well as I hoped

All four remaining participants were able to interpret the output of libFuzzer. Depending on the situation, they handled the corresponding situation differently.

CTF2-FS and CTF3-SF had problems with memory leaks due to how they implemented the fuzz targets. They were able to fix the problems and re-ran the fuzz target without the memory leak.

CTF7-SF wrote at least ten fuzz targets and ran them. Their fuzz target for the smb protocol crashed for every input. They decided that the fuzz target was flawed and just ignored it because they had several other fuzz targets that were up and running.

CTF7-SF's fuzz target for the dnp3 protocol also produced many errors, but again they understood that this was due to a flawed fuzz target and not because of actual bugs. They attributed the flaws to initialization problems and did not fix them for the same reason as before. CTF2-FS and CTF3-SF also had problems with initialization, but both fixed the issues to make the fuzz target work.

None of the four participants found a bug. However, all four were using libFuzzer correctly, and with more time available, it seems likely that they would have found the bug in the target program. While our pre-testing suggested that ten hours was enough time, future iterations of this kind of study should plan more time for this kind of task. Nonetheless, we are confident that they would be capable of finding these kinds of bugs with libFuzzer in the wild with the skill they already possess. However, the effort and skill required are quite substantial. In contrast, we do not believe that our CS would be able to use libFuzzer without investing significant effort in learning how to use the tool.

Expanding the Search As the participants did not encounter any true crashes, they felt the need of exploring further options. Most of them did this by manually targeting specific parts of the code. Still not encountering any crashes, they tried to optimize the fuzz targets and tried to develop more complex inputs to the functions. In the interviews participants CTF6-FS and CTF3-SF phrased this as a feature request.

Consequently, stateful fuzzing was needed. CTF3-SF considered implementing stateful fuzzing but was not able to do it in the given time. CTF6-FS implemented a minimal form of stateful fuzzing. However, they were not very enthusiastic about it:

> Libfuzzer does not support stateful fuzzing, therefore no high expectations as path stability will be horrible.

Corpus and Dictionary Except for CTF2-FS, all participants used corpora for their respective fuzz targets. Interestingly, CTF6-FS used both a corpus and a dictionary. CTF6-FS observed their fuzz targets with a corpus and a dictionary included and noticed a drop in performance because the coverage was lower than without the corpus and dictionary. Consequently, they proceeded without either.

4.7 Discussion

4.7.1 Clang Static Analyzer

The Clang Static Analyzer enabled even inexperienced users to check the target project for potential security issues. With the Clang Static Analyzer, both our participant groups were able to start the process reasonably easily and quickly. The usability of the tool was consequently viewed fairly positively. Our participants intuitively used Nielsen's view on usability [144], which separates usability and utility. In the hard task, the high number of false-positive warnings was seen negatively by both the CS and CTF groups, but this did not affect their perception of "usability". The CTF participants also had a negative view of the usefulness in general. They did not think the tool was helpful when looking for vulnerabilities. Consequently, they saw the tools as having good usability but bad utility. It is worth noting, though, that under the ISO 9241 [95] definition of usability, the bad effectiveness and efficiency measured against the capability of finding true bugs would lead the Clang Static Analyzer to receive a bad usability evaluation.

Thus, the holy grail of static analysis continues to be the reduction of the number of false positives. This would improve the utility under Nielsen or usability under ISO 9241 and enable users to effectively and efficiently find bugs.

4.7.2 libFuzzer

In stark contrast to the Clang Static Analyzer, where participants only struggled in the very last step, we found no step in the libFuzzer process that did not cause our participants severe problems. Our CS participants struggled even with the easy fuzzing task showing that the usability of libFuzzer is not at a comparable level to the Clang Static Analyzer. Even our skilled CTF players found many aspects vexing, unnecessarily complicated, and burdensome. However, in theory, the utility of libFuzzer is good. Consequently, we see a lot of potential if the usability can be improved.

Based on our observations, our recommendations for libFuzzer are:

- **Assist users in finding suitable functions to fuzz** It would be useful if libFuzzer assisted users in identifying functions worth fuzzing quickly. This was not an issue for our CTF participants, but if libFuzzer is to see the same level of adoption as static analysis, it needs to be usable by non-experts as well.

- **Fuzz-target creation** This is one of the most important points. It takes a lot of expertise to write anything but the most trivial fuzz targets for libFuzzer. In the case of Suricata, participants actually wrote multiple fuzz targets for the same function to account for the different parsers. Either assisting in creating fuzz targets or making the coverage-guided self-exploration of libFuzzer more intelligent would be a great benefit. It is essential for less experienced users, but it would also save time and effort for users like our CTF players.

- **Build automation** The building and linking process currently also requires a lot of manual work for non-toy projects, and it also requires a good understanding of how the different components interweave. It would be highly desirable to automate a lot of this so users do not need to understand, or know of, these issues.

- **Opt-out sanitizers:** Currently, the use of sanitizers is opt-in, i.e., the user has to integrate them actively. We would recommend including many of these by default and letting users opt out if necessary.

- **Support automatic stateful fuzzing** Many situations require stateful fuzzing to achieve good performance. In libFuzzer, this is a completely manual task, and some of our CTF participants even wrote their own stateful fuzzers to deal with the situation.

- **Improve Code Coverage** Our study shows that Code coverage plays a major role in the usability of libFuzzer. Even our CTF participants struggled to write fuzz targets that covered all the code of just one target function. This had to be done manually because libFuzzer is not yet powerful enough to do this on its own in a reasonable time. Potentially focusing on code coverage close to fuzz targets would be a worthwhile endeavor to increase usability.

- **Better documentation** Finally, while this is not particularly glamorous and is a well-known problem in many areas, we saw a clear need for better documentation. There is a clear difference between the Clang Static Analyzer and the libFuzzer documentation despite both belonging to the LLVM project. The current libFuzzer documentation led some of our participants astray. In particular, we recommend creating more complex examples instead of just using toy examples.

4.7.3 Comparison

Since we conducted a within-subjects study, we were also interested in our participants' comparative view of the two tools. To support our impressions from the interviews and diaries, we also analyzed the number of positive and negative comments to get an overview of the disposition towards the two tools.[2]

The majority of CS participants favored the Clang Static Analyzer when answering the question of which tool they would want to use in the future, including those faced with over 500 warnings in the hard task. In contrast, the CTF participants had a somewhat ambivalent relationship with the Clang Static Analyzer. In principle, they described the usability positively and had fewer negative comments for the Clang Static Analyzer than for the libFuzzer. However, they did not see the Clang Static Analyzer as a serious contender for finding vulnerabilities. As a result, they stated that they favored libFuzzer for future use and often stated that they would only use the Clang Static Analyzer for fixing style issues.

That is because they saw far more potential for libFuzzer than for the Clang Static Analyzer and thus would use libFuzzer. The corresponding Table A.7 can be found in the Appendix.

So, in summary, our interpretation of the results suggests that poor usability of libFuzzer and the good usability of the Clang Static Analyzer led CS students to prefer it despite the poor utility. However, the CTF participants acknowledged the better usability of the Clang Static Analyzer but saw too little utility to want to use it for their work in the future and tolerated the poor usability of libFuzzer due to its better-perceived utility.

4.8 Summary

In this paper, we presented the first qualitative studies examining the usability of libFuzzer and the Clang Static Analyzer. In the context of our study design, we found that the Clang Static Analyzer offers good usability but poor utility, while libFuzzer offers poor usability but better utility. Since static analysis and fuzzing find different kinds of bugs, ideally, they would both be used in tandem.

[2]The comment count does not necessarily reflect the weight of individual issues but offers interesting insights nonetheless.

For this, the usability of libFuzzer would need to be improved to lower the bar for entry. To aid in this, we identified several usability issues in libFuzzer and make suggestions for improvements.

Chapter 5

A Usability Evaluation of AFL and libFuzzer with CS Students

Disclaimer: *The contents of this chapter were previously published as part of the paper "A Usability Evaluation of AFL and libFuzzer with CS Students" at the ACM CHI Conference on Human Factors in Computing Systems (CHI). This was joined work together with my co-authors Mischa Meier and Matthew Smith. As this work was conducted with my co-authors as a team, this chapter will use the academic "we" to mirror this fact. I designed the study with advice from Matthew Smith. I set up and performed the study as well as designed and conducted all interviews. Mischa Meier supported me in the coding and the discussion of the interviews.*

Finally, before compiling the paper for publication, Matthew Smith and I jointly discussed the study's implications.

5.1 Motivation

Green and Smith called for more focus on the human aspects of software development [82]. Since then, many papers researching these aspects or improving the usability of software development have been published e.g. [139, 137, 136, 138, 47, 50, 49, 52, 85, 193, 89, 80, 191, 79, 70, 185, 186]. However, there are still many areas of software security where the human aspects have received little attention.

Fuzzing has emerged as a major hot topic in the security and programming languages research communities, with over 50 new fuzzing papers published over the last two years at the top four security conferences (IEEE S&P, ACM CCS, Usenix Security, NDSS) and ICSE [18, 62, 114, 141, 109, 86, 195, 130, 81, 124, 83, 196, 204, 92, 190, 194, 91, 205, 35, 160, 160, 65, 58, 87, 97, 152, 134, 208, 110, 115, 167, 133, 66, 59, 102, 104, 188, 199, 206, 68, 162, 148, 96, 31, 111, 71, 209, 151, 203, 122, 98, 154, 166, 15, 105, 189, 32, 57, 7, 90, 198, 153]. Furthermore, large software companies such as Google, Microsoft, and others use fuzzing extensively and effectively. Tens of thousands of bugs have been found with fuzzing [168], proving its worth.

However, the many software vulnerabilities that still get uncovered almost on a daily basis show that current countermeasures are not enough. We argue that one aspect of this problem is that fuzzing currently requires a high degree of expertise to use, and there are not enough experts to go around.

Concerning the expertise, Plöger et al. [156] conducted a qualitative study with CS students and Capture the Flag (CTF) players, they compared the usability of libFuzzer [117] and the Clang Static Analyzer [39]. In their study, the majority of participants had so much trouble with the fuzzer that in the end, only 3 of 32 students managed to run the fuzzer at all, and while four of the six CTF players managed to run the fuzzer, none of them found the bug.

Concerning the lack of experts, we did a back-of-the-envelope calculation using the professional network service LinkedIn [121] to estimate the ratio of regular software developers to security specialists. Our search turned up roughly 9 million people claiming developer expertise. Compared to that, we only found roughly 90000 IT security experts. This includes anybody claiming security expertise - not only those claiming software testing expertise and thus is a very generous upper bound. But even with this upper bound, the ratio of experts to developers is roughly 1 to 100. If we narrow the search to people mentioning fuzzing, we are down to only 2300 in total [1]. Despite this being only a back-of-the-envelope calculation, we think these search results clearly show that the ratio of fuzzing experts to developers is off by many orders of magnitude. The imbalance is backed up by a study by Wermke et al. [193], who showed that only five out of 27 owners, maintainers, and contributors from a diverse set of open-source projects were aware of a role in their projects that deals with security. So in many cases, security rests with developers who do not have specialized security expertise, with the predictable outcome we see in the number of software vulnerabilities. To echo Green and Smith, we do not blame the developer and do not think it a realistic or sensible plan to educate masses of developers to become security experts. Instead, we want to aid them by improving the usability of fuzzing so they can benefit from this powerful technology without becoming fuzzing experts. As a first step, we need to understand the current usability problems of fuzzers better. While the study conducted by Plöger et al. already highlights some issues, their study design, in combination with the very high drop-out rate, led to very little information being gathered on the latter fuzzing steps, such as actually running the fuzzer.

To remedy this and gather more fine-grained information on all steps of the fuzzing process, we designed an improved study protocol aimed at getting a larger sample into contact with all fuzzing steps. To accomplish this, we divided the monolithic fuzzing task from Plöger et al. into seven subtasks and provided an interactive study support system to assist the participants if they got stuck and to gather usability feedback in situ. We also extended the study to include the AFL fuzzer [4] as well as libFuzzer [117] to be able to compare two different fuzzing approaches with each other to see the strengths and weaknesses of each. We conducted the study within subjects with two 10-hour tasks - one for each

[1] Results may vary depending on region and time of search

fuzzer. We started the study with 47 CS students. Of those, 16 dropped out, but we gathered data on most steps from 31 participants. Twenty-five managed to run a fuzzer in at least one of the two tasks, and 17 completed both. While this is still a small sample size compared to end-user studies, it is the largest fuzzing study to date and offers a solid base to evaluate what causes problems and what already works well for two popular fuzzers. We highlight areas that work well in one fuzzer that are problematic in the other and vice versa. This offers insights into improvements that can be transferred from one fuzzer to the other.

Based on our findings, we suggest how the usability of AFL and libFuzzer can be improved to make fuzzing more accessible to non-experts and hopefully help even out the odds.

5.2 Methodology

To evaluate the usability issues of libFuzzer and AFL, we chose a within-subjects design so all participants would use both fuzzers.

5.2.1 Fuzzer Selection

We decided to limit our study to two fuzzers. While it would certainly be interesting to study more fuzzers, the length of the study and the difficulty of recruiting enough participants makes it infeasible to add more. Limiting ourselves to two fuzzers enabled us to use a within-subjects study design which is preferable for a small number of participants.

At the time of conducting the study, we wanted to use fuzzers that are popular in academia as well as industry. Klees et al. [106] evaluated 32 recent fuzzing papers and concluded that AFL was a very prominent fuzzer in the community as 14 out of the 32 investigated papers chose AFL as their comparison's baseline, followed by libFuzzer and zzuf [210], with 3 out of the 32. Moreover, Google's fuzzing projects OSS-fuzz [150] and Clusterfuzz [77] were prominent as they found several thousand bugs in open-source software [168]. Both were originally based on libFuzzer and have infrastructure additions to run on clusters, which was not relevant to our study. Therefore, we choose AFL and libFuzzer as our testing tools. Furthermore, since AFL is primarily an application-based fuzzer and libFuzzer a library fuzzer, it enabled us to compare the benefits and drawbacks of these fuzzer types as well. When we ran the study, AFL++ [5] was not yet as prominent as it now is.

5.2.2 Terminology

The documentation of AFL and libFuzzer sometimes refer to slightly different things with the same name or use different terms for the same things. In order to prevent ambiguity, we will give a short description of some important terms in the following.

- *target program:* We understand a target program as the software as a whole that a user of a fuzzer tries to fuzz. A target program can be, for example, a library or an executable. Participants have the source code and can compile and use the binaries.
- *fuzz target:* We understand a fuzz target as a piece of code that is written by the user that contains the code that runs the fuzzer on a specific function of the target program. In the case of libFuzzer, the fuzz target is usually the function *LLVMFuzzerTestOneInput*. For AFL, this is usually a stand-alone program with a separate main-function where a function of the target program is executed by the fuzzer.
- *target function:* We understand a target function as a function of the target program that is called by a fuzz target.

5.2.3 Target Program Selection

For our two target programs, we chose libroxml [119, 118] and tomlc99 [182]. Our choices will be explained in the following.

Target Program Difficulty

Plöger et al. had one easy and one hard fuzzing task in their study and compared their results between subjects. Their results showed that the hard fuzzing task was too hard to solve in the study's time frame, even for the competitive CTF players. So we decided to only use target programs that were fairly simple to fuzz.

We tried to find two bugs of similar difficulty by using target programs of comparable size and similar overall structure. We also tried to find bugs that could be found with similar effort on the fuzzing side. This has to be seen as a best effort though, as no two projects or bugs are alike.

We randomized the order of the target programs and fuzzers.

Bug Requirements

The selected target programs had to have a bug that could be found with fuzzing with little effort. While the bugs should be easy to find with the fuzzers, the bugs should be hard to find by other means. Consequently, participants should not be able to find a solution online, e.g., by finding hints in an issue tracker or using existing CVEs, and should not be able to find a solution by just browsing the code. Consequently, we did not use the same target programs as in Plöger et al. as these might already have solutions online. Like Plöger et al., we also did not want to use synthetic bugs, so we fuzzed open-source projects ourselves to find new vulnerabilities.

We are not aware of any sources that address the difficulty of bugs in fuzzing. Therefore, we have again followed Plöger et al. They considered a program to be easy to fuzz if it is relatively small, has only a small number of public interfaces, the necessary fuzz target is simple, the bug is found in a short period of time by the fuzzer, and no additional options or features are necessary. Even though these conditions are not very detailed, we agree with them and applied them according to our standards. We also choose to find target programs that parse files since we have had the same experience that these are usually perceived as easy to fuzz.

We then evaluated all the bugs we found and selected two programs based on our assessment of their suitability. To make sure that the bugs are found in a short period of time, we ran each fuzzer, with configurations setups if applicable, on the target programs 1000 times on the provided setup on a 5-year-old notebook. In every fuzzing run, a bug was found. For the slowest but still reasonable setup, fuzzing the afl-gcc-compiled slowest executable with AFL and activated ASAN, the average trigger time for a bug was about 4 minutes, and the maximum time it took to trigger a bug was 15 minutes.

Responsible Disclosure and Ethics

Since we were working with real vulnerabilities, responsible reporting guidelines were followed. We notified the developers of all programs about all found bugs righter after the study ended. Consequently, all participants had to agree to abide by the responsible disclosure practices. The entire experiment was conducted locally, and no risk to any live systems was posed.

The study complied with the EU GDPR and was reviewed and approved by the Research Ethics Board of our university.

libroxml

libroxml was one of the programs we selected. It is a library designed for parsing XML, which also contains two executables that both read in and parse an XML file to either resolve xpaths or to show the XML subtree. It contains typical functions for loading XML via a file or directly via a buffer. After loading XML, it can be used to navigate through the XML tree or access nodes. The project consists of 13 source files with about 7800 lines of C code. Therefore, it is nearly 10 times smaller than yaml-cpp which was used by Plöger et al. Moreover, the number of public interfaces is comparable. It uses Automake for the configuration and compilation.

Bugs We found one bug in libroxml suitable for our study. The bug could be triggered by loading an XML document via a file or buffer, arguably the library's most exposed function. As the bug could be triggered by loading an XML document it can also be found with the two existing executables. A straightforward fuzz target (FT) is sufficient to trigger the bug, i.e., no special seed, corpus, dictionary, or any form of sanitizer was necessary, and the bug could also be triggered without instrumenting the target program. More details on the bugs can be found in Section 3.1.1 and Appendix B.2.

tomlc99

tomlc99 was the second program we selected. It is a C library for parsing toml files. After reading in and parsing the toml file, it can also be used to traverse and locate a table in TOML and extract values. tomlc99 also contains two executables which again both read in and parse a toml file to either print the content to the console or transform it to JSON. With 1200 lines of code, it is about 6.5 times smaller than libroxml but still comparable because the overall structure of reading and parsing a file or buffer of a specific file format is similar. tomlc99 is built via a handwritten Makefile. We thought the simple Makefile might result in the build process of tomlc99 being easier to manipulate, but we did not see any evidence for that.

Bugs In tomlc99, we found one bug suitable for our study. The bug could be triggered by reading in a toml file via a buffer or a file. The bug could again also be triggered by using both executables. As in the case of libroxml, the functions from which the bug could be triggered are arguably the most exposed functions of the target program. As above, a straightforward fuzz target was sufficient, and no special seeds in the corpus or dictionary were needed. However, it was necessary to use an address or memory sanitizer. We did not think that the use of an address sanitizer would pose a problem and initially judged the two bugs to be of fairly similar difficulty. However, it turned out that it did cause more issues than expected, making the tomlc99 bug slightly more complex to find than the libroxml bug. A benefit of this was that we got additional insights into the important aspect of address sanitizers. Since fuzzer and program assignment were counter-balanced, the difference in difficulty should not affect the comparison of libFuzzer and AFL. More details on the bugs can be found in Appendix B.2.

5.2.4 Recruitment & Participants

Like Plöger et al., we recruited our participants from a usable security master's course in computer science.

We sent the pre-screening questionnaire to 129 students, of which 113 completed the questionnaire. Of those, we invited 59 students who reported having at least a basic proficiency in C/C++ and Linux and sent them a pre-questionnaire. 49 of the 59 invited students completed the pre-questionnaire and started the study. We excluded one participant (P31) as it became apparent that they did not have the minimum knowledge necessary and one (P07) because they worked as a student assistant in our group.

Demographics

To have a clearer picture of our group of participants, we asked them about their programming experience in years and programming proficiency in C/C++ on a scale from 1 very bad to 7 very good. Moreover, we asked them if they were familiar with the term fuzzing if they had used a fuzzer before, and if they had found a bug with a fuzzer. Due to a mistake, we lost the data on fuzzing experience from participants P46 and P48. It was striking that less than a handful of participants had used a fuzzer before and that 10 participants had 3 or fewer years of programming experience. In contrast, the proficiency in C/C++ revealed no major outliers. Neither participants with very low self-reported proficiency nor very high proficiency were part of our group. We will discuss possible implications in the later sections of this paper.

The overall demographics of our group can be seen in Table 5.1.

Gender	Male: 36	Female: 8	Other: 0	No Answer: 3
Age	min: 20, max: 35	mean: 24.46, median: 24	sd: 2.99	NA: 0
Prog. exp.	min: 1, max: 13	mean: 6.91, median: 7	sd: 3.37	NA: 1
Fam. fuzzing	Yes: 28	No: 17	NA: 2	
Used fuzzer	Yes: 5	No: 23		
Bug found	Yes: 4	No: 1		

Proficiency in C/C++									
Prof.	1	2	3	4	5	6	7	plot	median
Parts.	0	3	11	7	18	7	1		5

TABLE 5.1: Participant Demographics

Compensation

As in the study of Plöger et al., the participants were compensated with 10% bonus points for their end-of-term exam. Participants were only compensated if they completed all parts of the study. Students who did not want to participate in the study had other possibilities to acquire the same bonus.

5.2.5 Study Design

We used a within-study design since fuzzing is very skill-dependent, and individual differences can have a large impact. The assignment of the fuzzers, as well as the two target programs, were randomized and counterbalanced to roughly have all fuzzer and target program combinations equally distributed over the first and second tasks participants would work on.

As in Plöger et al., after filling out a pre-questionnaire, the participants had to work on two tasks consecutively. While working on the tasks, participants were asked to keep a diary about their work. In the end, a 45-minute semi-structured interview about their work on both tasks was conducted. The pre-questionnaire was based on the one of Plöger et al. It contained questions about demographics, programming experience, and previous knowledge about fuzzing. For the diary, participants received a based on the subtasks pre-structured document, and they were urgently asked in the task description to document their work, their bugs found, and the online resources used. However, the diaries differed greatly in their completeness and detail. The post-interview was again based on the one of Plöger et al. but was also structured based on the subtasks. Participants were asked, e.g., what they did in each step and their biggest problem for each step.

We also adopted the coding process of Plöger et al. Two researchers coded and analyzed the interviews using inductive coding [179]. For this, the interviews were randomly split into groups of three. The two researchers started coding the first three interviews independently and in parallel. The two resulting coding sets were compared, analyzed, discussed, and a new resulting set of codes was created. This new resulting coding set was used to again code the already coded interviews. Afterward, the subsequent three interviews were again coded in parallel and independently by the same two researchers with the current coding set. The coding sets were supplemented if codes emerged from the data in the process. Again the two resulting coding sets were compared and discussed, and a new coding set was created. The process for the last three interviews was repeated for the next three interviews until all interviews were coded. Therefore, in the end, every interview was coded with the resulting coding set. In this process, categories were developed, which were conceptualized into a model in further discussion. We then applied the themes included in the model to the qualitative analysis. So the qualitative analysis presents the categories and themes which were developed in the coding process.

All quotes from participants were translated from *redacted for review* to English by the authors.

The questions of the pre-questionnaire and semi-structured interview can be found in Appendix B.1 and B.3.

Plöger et al. offered participants pre-configured servers on which they could execute their fuzzing runs but reported that some participants struggled due to the lack of a GUI. To avoid this problem, we decided to offer pre-configured virtual machines that participants could run locally with a GUI.

All fuzzers and all necessary programs were pre-installed. The participants had root access to the virtual machines and were free to use any other software they wanted.

Time Frame and Limits

We kept the time limit and time frame for the tasks as in Plöger et al. Every participant had a hard time limit of ten hours to work on a task within a soft frame of ten days.

As far as possible, we encouraged participants to fully use the ten hours but we did not grant extensions to the number of hours they could work on the task. As with Plöger et al., we did however grant an extension to the ten-day time frame as long as the ten hours were not used up already. Participants were allowed to hand in earlier than ten days. Seven participants requested and got an extension for up to ten more days.

Task Structure

We decided to alter the task structure of Plöger et al. The task structure in Plöger et al. was monolithic, simply stating that the task was to find bugs with the given tool. Participants had to find out the necessary steps themselves and many participants got stuck and dropped out early, thus making information about later steps very scarce. To counter this, we divided the monolithic tasks into seven subtasks as outlined below. This gave participants some guidance and made the overall task easier. It also allowed us to implement a support system to help participants who were stuck on one subtask to move on to the next. While this approach allowed us to gather more usability data on all of the subtasks, which Plöger et al. could not, the support framework must be taken into account when interpreting the later steps.

The subtasks were derived from the fuzzing steps depicted in the companion document of Plöger et al.'s work. They are also a direct consequence of the steps described in the libFuzzer documentation [117], which are also identical to the steps needed to use AFL in the preferred persistent mode [129] and on libraries.

In the following, we present the subtasks from the task description. We stated that it was not mandatory for them to follow the subtasks, so if participants wanted to take a different route, they could. The seven subtasks were:

1. *Get a first impression of the target program and get an overall idea of what the program does.*
2. *Find a suitable function to fuzz.*
3. *Write your fuzz target in an external file.*
4. *Compile and instrument the target program.*
5. *Compile the fuzz target.*
6. *Run the fuzzer and interpret the output.*
7. *If necessary, adjust and improve.*

The final subtask was added to encourage the participants to dig deeper into the fuzzing process. From a technical point of view, these steps are not sequential. If we leave out subtask 1, which was implemented to provide a smoother entry into the task, subtask 4 could be done anywhere before subtask 5. However, we wanted to provide the participants with a clear structure to work with and thus decided to give them a simple list of steps. Furthermore, we decided against placing subtask 4 between subtasks 2 and 3, as those have a clear connection to each other. In the end, we settled with this order because the transition from subtask 1 to subtask 2 felt more in line than from subtask 1 to subtask 4.

Support System

The high drop-out rate in Plöger et al. hampered their ability to analyze the whole fuzzing process, as only three CS students made it to the end. To mitigate this problem, we implemented a support system similar to the one used in Tiefenau et al. [181].

Participants were told that if they got stuck they could contact support via e-mail. We stated that we would try to answer within a working day. We answered the first e-mail within 24 hours between nine a.m. and five p.m. but at the earliest after six hours. We implemented the six-hour rule because we did not want the support channel to be a quick way out. Follow-up questions on the same topic were answered as soon as possible, considering a typical working day and the working behavior of the study assistant, ensuring a normal conversation. Like Tiefenau et al. we divided our support into *hints* and *help*. A *hint* simply pointed the participant to the subtask, which was the cause of getting stuck, e.g. "Please look at the subtask: Compile and instrument the target program.". If that was not enough, we could follow up with a second *hint* which was more specific, e.g. "You need to instrument your program." If up to two *hints* did not enable the participant to get past the problem, we would offer actual *help* in solving the problem. If participants handed in their solution but were missing a subtask, we would apply the same support system.

For the sake of scoring the usability of the subtasks, we count any intervention as a fail for that subtask and all subsequent subtasks.

5.2.6 Scoring Systems

Successful Task Completion

We used the same method as Plöger et al. to determine a successful completion of the two tasks:

A participant successfully completed a task if the participant triggered a bug present in the code by using the respective fuzzer and recognized it as such.

Fuzzing Score

The fuzzing score awards a participant a point for every step successfully completed without support. We assigned a point for each of the following actions:

- Chose a useful function to fuzz
- Wrote a functional fuzz target
- Built the target program
- Instrumented all useful parts of the target program
- Built and instrumented the fuzz target
- Ran the fuzzer
- Used the address sanitizer
- Used a meaningful corpus or dictionary
- Triggered a bug
- Recognized a bug

Consequently, a participant was able to get a fuzzing score between 0 and 10. The steps reflect the necessary steps to fuzz a target program as well as the most fundamental optimizations in the form of the usage of the address sanitizer and a meaningful corpus or dictionary.

Since we did not restrict participants to following the subtasks, we had to accommodate alternative solutions in which some steps could be combined or skipped while still reaching the end goal. For example, a participant could decide to fuzz the executable of the target program with AFL instead of writing and using a fuzz target. If a participant successfully did that, they got multiple points. One point each for technically choosing a useful function to fuzz (the main function), have written a functional fuzz target (again the main function), built and instrumented all necessary parts of the target program, and also built and instrumented the fuzz target. Even if participants were not fully aware that this approach combined several steps, we consider the number of points fair since we believe this reflects good usability.

Exploratory Analysis

While we expected there to be differences between the two fuzzers, we had no basis for any kind of directional hypothesis and designed and ran our study as an exploratory mixed methods study. We gathered both quantitative and qualitative data from the participant's virtual machine, including the working directory and the bash history, their interaction with support, their diaries, and their exit interview.

For the comparison based on contingency tables, we used the odds ratio. In the case of central tendency comparisons, we report the difference between means. However, since this study is exploratory and we did not pre-define hypotheses, we do not place a large emphasis on statistical testing and do not report p-values. We do, however, report the 95% confidence intervals to highlight the expected range and uncertainty better. The confidence intervals for the odds ratios were calculated using bootstrapping. As described by Davison and Hinkley [55], this can cause the confidence intervals to be slightly smaller than the actual interval.

5.3 Limitations

Our study has the following limitations: In many ways, this study is a best-case scenario for the fuzzers. We chose moderately simple programs and broke the task down into subtasks. Participants also had the option of contacting us for support, which we logged and took into account in the evaluation. So the fairly poor usability we measured needs to be taken in this best-case context, and thus the real-world usability is likely to be even worse. In the following, we will go into more detail.

Participants

As it is very challenging to recruit professional developers for studies due to lack of time and high costs [1, 2, 171, 107], CS students are often used as proxies, and several studies have examined the validity of this practice [136, 137, 138, 3, 2].

Moreover, just recently, Tahaei and Vaniea [176] published their study on recruiting participants with programming skills. They recommended using CS mailing lists to recruit participants for studies where programming skills are required but also suggested collaborating with other universities to reduce validity issues. Moreover, Kaur et al. [103] compared six software developer samples to tackle the question of where to recruit for security development studies. They found that participants of all six recruiting platforms, including the student sample, report rich general software development and security experience, skills, and knowledge.

Thus, we also opted to use CS students for our study. However, it is conceivable that professional developers could achieve different results.

Unknown Code

Our participants were not familiar with the projects used in the study. Particularly the subtask for selecting a function to fuzz is likely to be different when fuzzing a project the participants are familiar with. However, this step was one of the few which caused very little trouble. Nevertheless, we think it is likely that developers of a company already have at least a rough overview of the codebase if they have to fuzz it and, thus, might perform differently.

Tools Selection

In our study, the participants only worked with the fuzzers AFL and libFuzzer. Although these fuzzers are widely used, other fuzzers might perform differently. Therefore, a generalization to other fuzzers and programming languages is not possible.

Project / Bug Selection

Since we did not want to work with synthetic bugs and needed two different bugs and projects for the within-subjects design, the bugs are not perfectly equal. Through randomization and counterbalancing, the effects should be distributed over the two fuzzers, but the choice of bugs and projects is still relevant to the overall outcome. Other bugs might lead to different results.

Guiding the Participants with Subtasks

By breaking down the overall task of finding bugs into more detailed subtasks, we gave participants more structure than they would have in the wild, where they would have to figure out the steps on their own. This was a trade-off to gather more data about later steps since Plöger et al. found that participants got stuck early on in the process. This means that our participants had an easier task than users would have in the real world. Consequently, fuzzing might even be more complex and challenging.

Support System

We offered our participants a support system. As with the subtasks, we introduced this system to avoid participants getting stuck or dropping out. For calculating the overall participants' scores, we discarded any points achieved due to the support system. So the participant scores could be lower than they would have scored without the support system because participants might feel tempted to choose the easier path of receiving support instead of solving the problem on their own. We saw no indications that make us believe this occurred. The benefit of the support system is that it helped participants reach deeper into the fuzzing process, which allowed us to gain more insights into all fuzzing steps. On a per-step evaluation, this was useful since it allowed us to assess the usability of a particular step of a fuzzer independently of previous steps. However, caution must be used when looking at how many participants worked on a specific step since, in this metric, the support system leads to a best-case scenario since participants might not have reached a specific step without help.

5.4 Results

5.4.1 Completion Rate and Success

Table 5.2 shows the overall drop-out and success rates for the two fuzzers. To be able to take ordering and learning effects into account, the results show the order in which participants used the fuzzer as well as the overall sum.

Apart from a slightly lower drop-out rate in the second task, there does not seem to be any great difference. In the overall score we can see that the drop-out and completion rates are fairly similar for AFL and libFuzzer (*OR* = 1.43, CI [0.51, 4.11]). However, almost half of the completed AFL studies ended in success while only a quarter of libFuzzers completions did (*OR* = 2.53, CI [0.68, 10.69]).

Overall, we could not identify demographic traits or skills, such as programming experience or proficiency, of our participants that impacted their completion rate or success.

Fuzzer	Order	started	drop-out	completed	success	
	1st	25	10	15	7	28%
AFL	2nd	11	4	7	3	30%
	$\sum$	35	13	22	10	29%
	$\sum$	37	17	20	5	14%
libFuzzer	2nd	15	5	10	2	13%
	1st	22	11	11	3	14%

TABLE 5.2: Overall drop-out, completion and success of our participants for AFL and libFuzzer.

The outcome for our 17 participants who completed both conditions and took part in the interview in terms of success can be seen in Table 5.3. Seven of 17 participants did not succeed with either fuzzer, and only 2 succeeded with both. But 7 who did not succeed with libFuzzer did succeed with AFL. In contrast, only 1 participant who did not succeed with AFL succeeded with libFuzzer. This time, the odds ratio was 7 and in favor of AFL (CI [1.33, 10]). So in the direct within-subjects comparison of the two fuzzers AFL clearly outperforms libFuzzer. On average, the odds of a participant succeeding with AFL are 7 times higher than with libFuzzer.

		libFuzzer		
		success	no success	∑
AFL	success	2	7	9
	no success	1	7	8
	∑	3	14	17

TABLE 5.3: Contingency table of outcome for AFL and libFuzzer in terms of success.

Target Program Results

We also compared the results of the two projects. As can be seen in Table 5.4, nearly three times more participants successfully solved the fuzzing task for libroxml than for tomlc99 (*OR* = 3.46, CI [0.89, 16.72]). This is interesting since tomlc99 is the smaller project with the arguably more straightforward compilation method. But, the need to use an address sanitizer for tomlc99 proved to be more of a challenge than we expected. Due to the counterbalanced nature of our study design, both fuzzers were affected by this fairly equally, so we do not think this is an issue for the evaluation.

Target Program	started	drop-out	completed	success	
libroxml	36	15	21	11	31%
tomlc99	36	15	21	4	11%

TABLE 5.4: Overall drop-out, completion and success of our participants for the target programs.

Additional Bug Found by Participant

Participant P15 found a bug in tomlc99 using AFL that was not known to us beforehand. P15 provided several crashing inputs with which we were able to confirm the bug. P15 did not find the study bug, but was still rated as successful as they had found a true bug. The bug was reported to the tomlc99 project.

5.4.2 Fuzzing Score Results

For all participants, the median fuzzing score, as described in Section 5.2.6, of AFL is 8 compared to 6 for libFuzzer, and the mean score of AFL is 7.08 compared to 5.81 for libFuzzer.

If we only consider the 17 participants that finished the study, the differences between AFL and libFuzzer are stronger. For those 17 participants, the median score of AFL is 9, and the mean is 8.24, while the median and mean in the case of libFuzzer are 6 and 6.65. Therefore, the difference between means is 1.59 (CI

[0.71, 2.35]). On average, the groups are divided by more than one and a half points in favor of AFL.

The differences in scores between AFL and libFuzzer for the 17 participants who finished the study become even more apparent when we look at the separate steps as shown in Table 5.5.

Each cell contains a ratio and a percentage. The denominator of the ratio is the number of participants that worked on a step, while the numerator is the number of participants that were successful in it. The percentage is the percentage of this ratio. This gives a usability assessment of each step in isolation and can highlight where usability improvements would be most needed, e.g., libFuzzer would benefit from helping users instrument the target program while AFL does this very well already. Both fuzzers could benefit from better usability for ASAN, although here libFuzzer fares better than AFL. However, in most steps, it can be seen that libFuzzer lags behind AFL.

Fuzzing Steps	AFL		libFuzzer	
Selected function	16/17	94%	16/17	94%
Wrote correct fuzz target	16/17	94%	8/17	47%
Built target program	17/17	100%	17/17	100%
Instrumented target program	17/17	100%	9/17	53%
Built fuzz target	17/17	100%	17/17	100%
Ran fuzzer	16/17	94%	16/17	94%
Used ASAN	5/6	83%	9/9	100%
Used Corpus/Dict	16/17	94%	13/13	100%
Triggered bug	12/16	75%	5/16	31%
Correctly interpreted bug	9/12	75%	3/5	60%

TABLE 5.5: Ratio and success percentage of steps by fuzzer.

A detailed listing of how our participants performed in every step of the fuzzing process and for every step of the fuzzing score can be found in Appendix B.4. There it can be seen that with the exception of two participants, every participant who finished both parts of the study reached every subtask. In comparison, in Plöger et al., only two of the six participants who finished the task got to run the fuzzer, and two participants did not even reach the step of building and instrumenting the target program.

Thus, our modified study design with the subtask outline and support system allowed us to gather significantly more data on all subtasks, leading to a richer understanding of the usability issues. However, in particular the subtasks make life easier for our participants than in the real world, and thus the poor results would likely be even worse in a real-world setting.

5.4.3 Analysis of Participants' Traits and Performance

While our study did not set out to measure the effect of programming skills on fuzzing, we did gather some data on participants' programming skills and fuzzing experience. To gain insights for future work we conducted a post-hoc exploratory analysis of this data. For all correlation tests with two ordinal variables or an ordinal variable and a binary variable we used Kendall's rank correlation test. In the case that two binary variables were present, Pearson's correlation test was used. The 15 correlation test results can be seen in Table 5.6. We want to stress that these tests were conducted in an exploratory manner and thus false positives are likely. Thus, the results should only be used to motivate future research. We see a potential small trend that libFuzzer benefits from more experience, which would be in line with our qualitative finding that it is the more complex solution. However, on the whole, we see that most effect sizes are fairly small and not statistically significant, and thus previous experience does not seem to have played a major role in our study.

independent variable	dependent variable	fuzzer	p-value	effect size
prog. proficiency	score	AFL	0.79	τ_b = -0.05, CI [-0.50, 0.39]
		libFuzzer	0.06	τ_b = 0.38, CI [-0.01, 0.78]
	finished		0.97	τ_b = 0.00, CI [-0.28, 0.29]
	success	AFL	0.56	τ_b = -0.08, CI [-0.37, 0.21]
		libFuzzer	0.02	τ_b = 0.30, CI [0.01, 0.62]
prog .experience	score	AFL	0.86	τ_b = 0.03, CI [-0.42, 0.48]
		libFuzzer	0.69	τ_b = 0.08, CI [-0.41, 0.56]
	finished		0.07	τ_b = 0.23, CI [0.01, 0.46]
	success	AFL	0.21	τ_b = 0.16, CI [-0.04, 0.35]
		libFuzzer	0.12	τ_b = 0.20, CI [-0.06, 0.44]
used fuzzer before	score	AFL	0.40	τ_b = 0.19, CI [-0.15, 0.54]
		libFuzzer	0.69	τ_b = -0.09, CI [-0.56, 0.39]
	finished		0.34	ρ = 0.17, CI [-0.12, 0.44]
	success	AFL	0.24	ρ = 0.18, CI [-0.11, 0.45]
		libFuzzer	0.29	ρ = 0.19, CI [-0.10, 0.45]

TABLE 5.6: Correlations effect sizes of participants' traits and performance via Kendall's rank correlation and Pearson's correlation test.

5.4.4 User Rating of Fuzzers

At the end of the study, we asked participants to rate the usability of the fuzzers and also give an overall score both on a scale from 1 (very bad) to 7 (very good), the corresponding Table can be found in Appendix B.5. For the overall rating, the mean and median for AFL were 5.18 and 5, while for libFuzzer they were 4.29 and 4. This results in a mean difference of 0.88 (CI [-0.04, 1.81]) in favor of AFL.

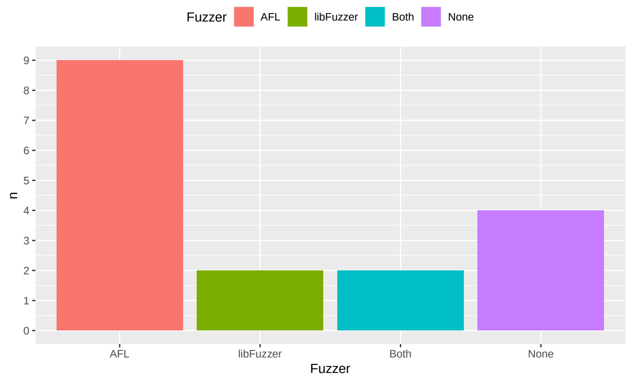


FIGURE 5.1: Number of participants that intended to use AFL or libFuzzer in the future.

Surprisingly, this was not reflected in the Usability score. Despite all other comparisons favoring AFL, the mean usability scores are almost identical, and the mean difference confidence interval is spread equally over the two fuzzers. The mean and median for AFL were 4.53 and 5, while for libFuzzer, they were 4.47 and 5. Thus, the difference between means is very small (*MD* = .06, CI [-1.21, 1.32]), giving no fuzzer an edge. The median value of 5 represents an evaluation of "somewhat good". This might seem surprising considering the many problems the fuzzers have. However, the survivors' bias must be taken into account when interpreting these numbers, i.e., participants who dropped out of the study did not rate this question, and thus the 5 is likely an optimistic upper bound.

However, we also asked the participants if they wanted to use one, both, or none of the fuzzers, in the future. As depicted in Figure 5.1, the results show a clear preference for AFL. Although, this also needs to be taken in the context of the 64% drop-out rate where participants chose to use neither fuzzer.

5.4.5 Support System Insights

18 of the 47 participants received support for at least one step in one of the two tasks. Ten of the 18 Participants dropped out in one of the tasks despite receiving support. Of the 18 participants who finished both tasks eight received *hints*.

In total, support was given 41 times, with 38 being *hints* and three being *help*.

An overview of the distribution of all given *hints* for both tasks can be seen in Figure 5.2. The most striking differences between AFL and libFuzzer are the instrumentation of the target program and the construction of the fuzz target. It also stands out that our participants often needed assistance with ASAN for both fuzzers. We will discuss this in more detail in our later analysis.

The initial support delay of 6 hours before responding to a request for support led to one participant solving their problem on their own after all.

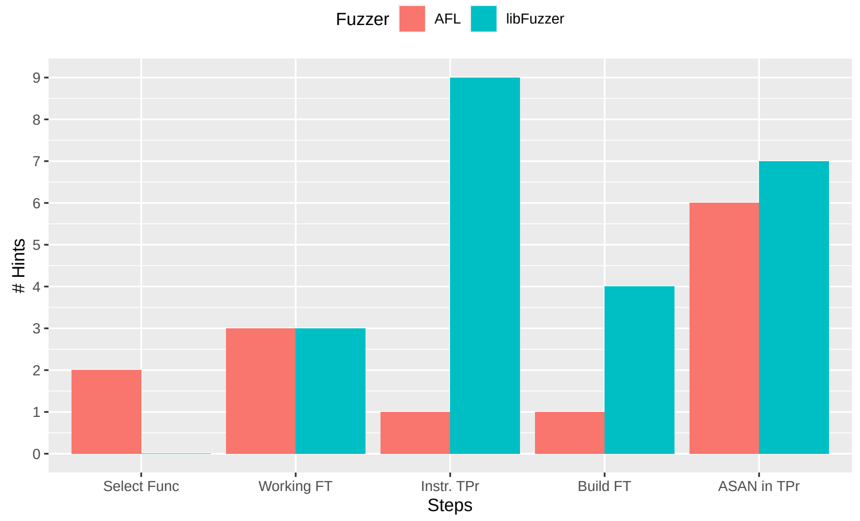


FIGURE 5.2: Number of *hints* given per steps for both tasks combined.

Both times the *help* was given it was in the first condition, once when the participant was using AFL and once when the participant was using libFuzzer. The *help* was given for building the fuzz target and starting the fuzzer. Both participants who received *help* dropped out of the study. One participant dropped out directly, while the other participant dropped out in the next condition.

While the support system did not prevent as many drop-outs as we had hoped, it was still very beneficial to us since it gave us deeper insights into where participants had problems.

In the following, we will report our observations from the analysis of the interviews, logbooks, and support interactions to give more detailed insights into the different steps, with the aim of offering a basis with which to improve the fuzzers.

From the data, we could not draw connections between traits and skills, and performance. However, the qualitative analysis, especially from the interviews, revealed a tendency that perseverance and a high tolerance for frustration were factors for success.

5.4.6 Familiarization with the Fuzzers

To familiarize themselves with the fuzzers, each participant read the documentation for each fuzzer linked in the task description. For AFL, more was needed to understand the fuzzer for most of our participants, and they started to google AFL and its usage. Consequently, participants frequently expressed frustration with the documentation in the interviews.

> The documentation was awful. (P16)

In the interviews and diaries, our participants reported that in the process, they came across several websites, which they searched through to gather all the information that was sufficient for them eventually. Neither in the interviews nor the diaries were the same websites mentioned by multiple participants in the case of AFL.

Our participants viewed the documentation of libFuzzer less negatively. Nevertheless, they explored other sources to get information. In their search for other sources, it was striking that many participants mentioned the google fuzzing tutorials [78] as a useful source.

Overall, participants strongly focused on written information. Other sources, such as YouTube videos, were only mentioned once.

5.4.7 How did Participants Select a Function to Fuzz

We asked participants how well they thought they had managed to get an overview of the target program. They mostly reported that they had gotten a good overview with a median score of 5 out of 7.

They mostly felt they could make an informed decision about which function to fuzz. Most participants expressed that they chose a specific function to fuzz because it was directly exposed in the API of the target program. It was a high-level function concerning the calling hierarchy or the first entry point for users.

5.4.8 Fuzz-Target-Less Fuzzing with AFL

Eight participants used Fuzz-Target-Less (FTL) fuzzing. In the case of FTL fuzzing, participants had to compile the target program the same way as when writing a fuzz target but did not have to write and build the fuzz target, which could make the task easier. Participants mentioned different reasons for doing this. Participants P08, P12, P14, and P44 used FTL fuzzing because they believed this would be the easiest way of getting the fuzzer to run. Participant P12 specifically mentioned that they intended to get back to writing the fuzz target if this method did not turn out to be fruitful. Since they were able to trigger a bug they stayed with this approach. Participant P26 and P48 were were struggling with the task and found FTL instructions in the documentation. Participant P06 misinterpreted the task thinking they were asked to fuzz the functionality of the whole target program instead of a function of the target program. None of the seven participants had an accurate picture of which part of the target program the fuzzer reached. P08 stated:

> I mean, when I call the program, the main function is executed. That means it has to go through that somehow.

In contrast to the other seven participants, P15 actively chose a function to fuzz, which was the parsing function in their case. However, they then realized that the parsing is triggered within the main function of the target program and thus decided to use FTL.

5.4.9 Writing the Fuzz Target

Of the eight participants that wrote a fuzz target for AFL, none used the recommendable persistent mode. Instead, they wrote a main function without the signature AFL loop, resulting in poor performance. While it was sufficient for our simple bug, this is not ideal. A recommended improvement for AFL is that it should inform users of this mode and assists in using it.

With libFuzzer, participants recognized the need for the LLVMFuzzerTestOneInput function and how it should be used. However, they often struggled to correctly pass the Data argument of the LLVMFuzzerTestOneInput function to the target function since the target function expected a string and not a pointer to a character or character array. Passing a pointer to a character or character array instead of a string led to the problem that the passed element was not null-terminated. Nonetheless, those fuzz targets could be compiled without errors. However, the non-null-termination resulted in many crashes in conjunction with an address sanitizer. Critically, most participants had trouble figuring out the cause of the crashes, leading to delays and frustration.

This problem did not arise for AFL because inputs are null-terminated by default.

5.4.10 Building and Instrumenting

In the case of AFL, the compilation and instrumentation of the target program and the fuzz target revealed minor problems. One was that participants already compiled the target program with the standard compiler while making themselves familiar with the target program and thus overlooked the compilation of the target program with one of the compilers of AFL. However, they found out about the problem and could fix it on their own. The error messages AFL showed when starting the fuzzer were considered very helpful, e.g., participant P08 stated:

> But then I had the error stating that the binary was not properly instrumented again [...] and then I recompiled the program, and then the code was properly instrumented.

On the other hand, participants had more noteworthy problems in the compilation and instrumentation process when using libFuzzer. Five of the participants reported that it was not apparent that the target program should be instrumented. Participant P08 also said:

> Yeah, I didn't really pay attention to that back then. I think I only noticed that in the second task. But the program has to be instrumented

> so that the fuzzer can work with it. It actually needs to be compiled in a certain way and not simply with the standard compilation that one might otherwise use. But then I didn't actually do that since I only used make.

Three other participants also did not instrument the target program. Reasons for that were that instrumentation did not look important or that the participants thought they would also instrument the target program when building the fuzz target. Participant P44 stated:

> The target program could be compiled without any special fuzzing attributes. But the fuzzing target must be compiled with the fuzzer. And if you link the two together, then the program should have been instrumented.

Most participants had problems understanding the connection of the target program, the fuzz target, instrumentation, linking, the compiler, and the fuzzer. The problems were especially apparent when participants had to work with libFuzzer. The participants had problems differentiating between the target program and the fuzz target and did not understand what instrumentation is and what needs to be instrumented. Moreover, participants had problems linking the target program with the fuzz target and choosing which compiler to use. Furthermore, they struggled to understand what parts of the process are influenced by the fuzzer. The questions about instrumentation and how to link the target program with the fuzz target stood out. The problem of correctly linking the target program with the fuzz target frequently resulted in a guessing game where participants tried to find the right combination of things like CC, make, CFLAGS, configure, afl-gcc, clang, and fsanitize. As a last resort, some participants used make install, which helped them find the right arguments to compile the fuzz target.

Overall, participants perceived the compilation and instrumentation as time-consuming and tedious.

5.4.11 Running the Fuzzer

Several of the participants had problems starting AFL. The problems were mainly based on passing the input of the fuzzer to the executable. The option of giving the executable a file as input or giving the input to the executable via the standard input stream was confusing. The confusion was not based on the two possibilities of passing the input to the fuzzer, but on how to pass those inputs to the fuzzer. The participants understood the concept of the corpus directory. However, they struggled to understand when the path or content of a file was given as input. In this context, participants had problems grasping the "@@" option. It was hard for the participants to understand that the starting command of the fuzzer included the path of the corpus in both cases but that the "@@" option determined the form of input. The participants were warned by AFL with the comment "odd, check syntax!" in the UI. They recognized the warning, but had problems to identify the cause. In the end, they experimented, until the problem was solved.

Moreover, AFL warned every one of our participants via an error message when starting the fuzzer that a modification of the OS core pattern is needed to avoid crashes being misinterpreted as timeouts. This was seen as disruptive and tedious. However, the error was informative so that fixing it was not a problem for the participants. Participants praised the error messages AFL showed here. An example of such an error message can be seen in Appendix B.6.

This error message is only shown when FTL fuzzing was used. It is not shown in the case that a fuzz target was written and compiled and linked to a non-instrumented target program. Participant P16 was the only participant to encounter this scenario and was only warned by AFL with the comment "odd, check syntax!".

In the case of libFuzzer starting the fuzzer was seen as an uncomplicated step. The participants praised the simplicity of executing the fuzz target and were pleased with running the fuzzer.

5.4.12 Run-time Output

The output while running the fuzzer was often rated very positively by the participants for AFL. In contrast, the output of libFuzzer only received negative comments. Participant P12 summarized this as follows:

> I noticed positively that AFL generates much nicer, much easier to understand statistics while the [AFL] is running [compared to libFuzzer].

Examples of the output of both fuzzers can be found in Appendix B.7.

Fuzzing Metrics

When using AFL, participants had a variety of metrics to determine whether the fuzzer was running well or not. Unsurprisingly, as they appear in bright red, the appearance of crashes was the most named reason for a good fuzzing run. P15 stated:

> I was sure when we found the first crash because a nice red number appears, and it tells you where [AFL] stored it.

Moreover, the number of paths detected and the increase of the number of paths was also seen as a good measurement but partially hard to interpret. Participants reported this was the case because they could not tell what magnitude for the paths was necessary to have good coverage, an acceptable coverage, or poor coverage. Participants also sometimes based the success of their fuzzing on the growth and appropriateness of the corpus.

For libFuzzer, a crash was also by far the most named indicator with which to tell if the fuzzer is running well because the fuzzer stops when it finds a crash. As with AFL, the number of paths was also the second-best indicator. However, participants struggled to tell whether the fuzzer ran well with libFuzzer more often compared to AFL. Most of the time, they decided that the fuzzing process was running fine since it was generating some outputs. P06 said:

> ... the program wrote something on the command line every few seconds. So it seemed like it was working, even if I didn't know exactly what it was doing.

5.4.13 Crash Output

In contrast, participants' ratings of the crash output of the fuzzers were reversed.For the crashes, the output of AFL was not rated positively even once, while some participants gave negative feedback about it. In contrast, the output for crashes of libFuzzer was partially seen positively and rarely was given negative feedback.

Our participants mentioned several reasons for this. One reason was that, even though AFL gave direct feedback in the interface that a crash occurred, additional information had to be searched for in the output folder. This was seen as tedious and some participants were not able to find it.

libFuzzer was helped by the fact, that many participants used the address sanitizer, which augments the output with additional information. Participant P06 stated on the matter:

> Well, after it threw an error, the output was pretty helpful because I could just see right away where to look next. It also told me what type of error occurred. That was helpful.

Participants were pleased with getting a name of the function and a line number of where the crash occurred.

Crash Check

Three participants said that they analyzed the crashes by manually browsing the code and looking at the respective positions in the code. Only four participants validated crashes by running the executable of the target program with the crash file. Those four participants valued this method of confirmation since they perceived it as very safe. One of the four participants ran the executable in conjunction with Valgrind [140] to get further information on the crash. The method most commonly used was a debugger such as vanilla GDB [75] or GDB with the pwndbg [158] plugin. Seven participants in nine tasks used a debugger. However, the certainty about a crash being a bug after the inspection with a debugger varied considerably.

We perceived that participants had noticeably higher confidence of crashes being actual bugs because the crashes were produced with the executable provided by the target program and not via a self-written code snippet.

When to End the Fuzzing Campaign

Participants faced the problem of deciding when to stop the fuzzing process when no bugs were found. The documentation of AFL suggests a combination of different approaches to this problem. Broadly speaking, proper coverage in combination with a green-colored cycle count is a good indication for stopping the fuzzer. Three participants were aware of the meaning and the purpose of the cycles, but only participant P19 reached the point where the paths were reasonable, and no crashes were found. Nonetheless, participant P19 stopped the fuzzer before the cycles were green because they felt that the paths were already very high and no significant improvements were realistic.

In the libFuzzer documentation, no explicit recommendations are given regarding the problem when to stop the fuzzer. Although, it is recommended to use coverage visualization to check if improvements can be made to the fuzzing process. We will discuss coverage visualization in the Section improvements to the fuzzing process.

Overall, most participants followed the idea of stopping the fuzzer when they had the feeling that nothing new would happen. Participant P25 summarized it as follows:

> Then I thought, okay, how long do I let this run now? Because normally, you should always let a fuzzer run for a longer period of time. However, I then noticed relatively quickly that after thirty minutes, nothing new had happened. Then I let it run for another hour and then I stopped it.

5.4.14 Improvements to the Fuzzing Process

In the last subtask, participants were asked to improve the fuzzing process. We did not specify how because we were interested in seeing what they would find and deem promising. As a reminder, the bug in libroxml could be found without this subtask, whereas the bug in tomlc99 required the use of the address sanitizer, which falls into this subtask. In the following, we set the focus on the most important improvements chosen. More information on other improvements can be found in Appendix B.8.

Corpus and Dictionary

A popular improvement that participants chose for both fuzzer was to work with meaningful seeds in the corpus. For both fuzzers, this was not an issue for the participants. While AFL must have a corpus right at the beginning, it was not uncommon for participants to start with a very minimal corpus, for example, just one random file, but to adjust the seeds in different ways later. The creation and adjustment of meaningful seeds were made in different ways. Some participants searched for existing corpora for the specific parser on the internet, while others created the seeds by hand. Other participants used the existing test cases in the projects. Usually, all groups had in mind that they wanted to have a corpus containing valid and invalid inputs for the target.

Nearly all participants ignored the possibility of using a dictionary. Only three participants used them, mainly because it was just another thing in the documentation, and they wanted to explore this option.

Sanitizers

The address sanitizer was the by far most used sanitizer as it was one of the few mentioned in the documentation.

A positive aspect of AFL's address sanitizer was that if the sanitizer was used, the fuzz target, the target function, and all other necessary functions were instrumented with it. On the downside, the ASN documentation of AFL was hard to understand and received negative feedback, e.g., participant P14 said:

> When I started to use the address sanitizer, it found a crash quite quickly. But in the beginning, it was definitely not obvious that [ASN was useful].

With libFuzzer, participants used more kinds of sanitizers but the address sanitizer was still the most common and was perceived as the most useful. In contrast to AFL, participants had no problems finding the right commands to use the sanitizers with libFuzzer but often only applied it to the fuzz target and not the target program. Frequently, participants were unaware that the target program should also be compiled with sanitizers or did not properly understand the process.

Coverage Visualization

In AFL, the code coverage visualization is offered via a third-party tool. Whereas libFuzzer requires two compiler arguments and two executions of different programs. As a result, none of the participants looked at the coverage when using AFL, and only two participants tried to use the visualization provided by libFuzzer but failed to do so. This also meant that participants struggled to determine whether the fuzzer was doing something meaningful.

Consequently, participants were also unsure when to end the fuzzing campaign.

5.5 Discussion

The overall success rate for all 47 participants was 29% for AFL and 14% for libFuzzer. If we only consider the 17 participants who finished both tasks, the success rates were 53% for AFL and 18% for libFuzzer. Our mixed methods analysis shows AFL having better usability than libFuzzer in many - but not all - areas. But even AFL did not offer what we would call good usability.

While giving participants a more structured task and a support system resulted in substantially more participants reaching all steps compared to Plöger et al.'s study, we still had a substantial number of participants who did not finish the entire study: 30 from 47 participants. This highlights that fuzzing - even with a fairly simple target program - is a tough challenge for CS students in a 10-hour time frame. 64% of our participants dropped out, and even some who made it all the way through stated that they would not want to use fuzzers again. Although often overcoming arising problems, it is noteworthy that even the final 17 participants struggled with nearly every step of the fuzzing process. This really highlights that it is important to invest research effort into the HCI of fuzzing as well as into new features.

In our estimation, fuzzers are currently products of cutting-edge research and are built by experts and used by experts who are willing to invest the time and effort needed. However, this makes the entry barrier very high for regular programmers who would benefit from fuzzing their code but do not have the time or inclination to become fuzzing experts. The results presented in the previous section show that large usability improvements could be made. First, we add weight to some as well as question some recommendations made by Plöger et al. then we present further recommendations based on our observations.

5.5.1 Extensions

Better documentation

Unsurprisingly, like Plöger et al., we also see a clear need for a better documentation for both fuzzers. We can strengthen this recommendation by our observation that the simple list of steps we gave participants as part of their task description was already a great help to our participants. Consequently, we would recommend that the documentation also add a high-level overview instead of only focusing on the details since we saw that many participants did not get how the steps interact and how they build on each other.

Build automation

We also agree with Plöger et al.'s finding that automating the building process for the target program as well as the fuzz targets can be very beneficial for the users. The building and instrumenting of both were a major pain point for many of our participants, mostly for libFuzzer but also for AFL.

Opt-out sanitizers

While we also observed that it would be very beneficial if more participants had used sanitizers, we are not sure whether the opt-out sanitizers suggested by Plöger et al. are a good idea. In our larger sample, we saw that many of our participants already struggled with the complexity of the fuzzing process and did not have a good understanding of what was going on. So adding a large number of extra sanitizers and requiring users to opt-out could be detrimental to some users. Instead, it might be a better option if fuzzers could guide users in the use of extra sanitizers (see 5.5.2).

5.5.2 New Recommendations

Error Messages

The error messages offered by AFL were generally rated favorably by our participants, while libFuzzer warnings were considered unhelpful or even non-existent. For example, libFuzzer does not warn participants if the target program was not instrumented, causing many problems for our participants. Improving libFuzzer warnings and feedback based on AFL is thus a simple yet effective solution.

UI Guidance

But more generally, it would be beneficial if the UI guided the user through the different steps proactively. We saw that the simple list of steps we gave our participants helped them compared to Plöger et al.'s task description. Integrating these steps into an interactive UI could help novice users. Currently, many important steps, such as using ASAN, must be known a priori because the UI does not point them out or integrate them automatically. The same goes for feedback about how the fuzzing run is going. Participants often stated that they did not know if the fuzzer was working correctly, or worse yet, thought that it was even though it was not. Furthermore, getting graphical feedback during run-time is possible but cumbersome. We would suggest graphical feedback should be the default option that experts can opt out of if they want to.

Metrics of Fuzzing Runs

The metrics about fuzzing runs currently offered by the fuzzers did not help our participants to judge whether the fuzzer was running well or when it was OK to end a fuzzing run. Developing more helpful explanations for current fuzzing metrics or potentially more user-friendly metrics themselves is an open research challenge.

Crash Analysis

Helping users understand and analyze crashes automatically would also be a substantial help. For our scenario, it would have been particularly useful if libFuzzer had been able to recognize that many of the crashes were caused by problems in the fuzz target (UEICs). However, it would be even better if UEICs were not an issue at all because fuzz targets are generated automatically.

Fuzz target-less fuzzing

AFL's ability to fuzz an executable without the need to write a fuzz target was a great help, but it only works for programs that receive their controlling input from standard-in or command-line arguments. It would be very desirable for fuzzers to be able to automatically fuzz arbitrary functions without much input or fuzzing knowledge required from the user. This is, of course, an open research challenge, but one which is very worthwhile both from a technical as well as an HCI perspective.

5.6 Summary

In this paper, we present the results of our 20-hour user study with CS students comparing the usability of the two popular fuzzers AFL and libFuzzer.

We found that our participants got better results when using AFL compared to libFuzzer, but both fuzzers were a challenge for our participants. If fuzzing is to realize its full potential and be used by regular developers who are not fuzzing experts, the usability needs to be improved significantly. Our analysis highlights which steps were particularly challenging and where usability improvements are necessary. Fuzzing is an incredibly effective tool in the right hands, but usability issues currently limit the impact outside of research and the top-tier companies capable of hiring fuzzing experts. Thus, focusing on the usability issues raised in this paper and designing and testing improvements is recommendable future work that can have a significant impact on the wide-scale adoption of fuzzing.

Chapter 6

Comparison of AFL and libFuzzer with Freelance Developers

Disclaimer: *The content of this chapter was joined work together with my supervisor Matthew Smith. As this work was conducted with my supervisor as a team, this chapter will use the academic "we" to mirror this fact. I designed the study with advice from Matthew Smith. I set up and performed the study as well as designed and conducted all interviews.*

Finally, before compiling the paper for publication, Matthew Smith and I jointly discussed the study's implications.

6.1 Motivation

Fuzzing continuose to be a hot topic in the security and programming languages research communities, with over 50 new fuzzing papers published over the last two years at the top security and software engineering conferences (IEEE S&P, ACM CCS, Usenix Security, NDSS and ICSE) [18, 62, 114, 141, 109, 86, 195, 130, 81, 124, 83, 196, 204, 92, 190, 194, 91, 205, 35, 160, 160, 65, 58, 87, 97, 152, 134, 208, 110, 115, 167, 133, 66, 59, 102, 104, 188, 199, 209, 68, 162, 148, 96, 31, 111, 71, 151, 203, 122, 98, 154, 166, 15, 105, 189, 32, 57, 7, 90, 198, 153]. However, these papers focus heavily on the technical aspects of fuzzing, such as code coverage or opening up a new system to fuzzing. The usability side of fuzzing has received far less attention. Plöger et al. have published two studies with CS students in this domain. In their first study, Plöger et al. [156] conducted a qualitative analysis of libFuzzer and the Clang Static Analyzer. They found that their participants saw greater potential with fuzzing; however, the usability was sorely lacking, with only 3 of 32 students managing to run the fuzzer. In their second study, Plöger et al. [157] conducted a fuzzing study with 17 CS students to evaluate the usability of two prominent fuzzers AFL [4] and libFuzzer [117]. In the study, participants were asked to find security-related bugs in two different software projects - once using AFL and once using libFuzzer. The study was conducted within subjects, and the students had ten hours per fuzzer for a total of 20 hours in the study. The main finding was that while AFL performed better both in finding bugs and in the subjective usability evaluation of the participants, students struggled with both fuzzers, and there were many areas for improvement.

However, the results still need to be taken with some caution since the sample size was small and the fuzzers were only tested by CS students. The small sample size limited the statistical power, and the student sample might limit the applicability of the results to other types of developers. To address both these limitations, we conducted a power analysis to determine how many participants would be needed for a sound statistical analysis and based on that, we hired 150 freelance developers to take part in a replication of Plöger et al.'s study, of which 62 completed the study. We compensated freelancers with 430€ for their effort. Our results show that despite having a mean of 10 years of experience, freelancers struggled with many of the same issues that the CS students struggled with and actually had more problems with the usability of the fuzzers than the students. Based on our results, we highlight similarities and difference between the student and freelancer samples and highlights the need for usability improvements if fuzzing is to see more widespread adoption outside of academia and top-tier tech companies such as Google, Microsoft, etc., that have dedicated fuzzing departments.

6.2 Ethics & Private Disclosure

Since one of the vulnerabilities was not yet fixed when replicating the study, we had our participants commit to a private disclosure agreement before being allowed to enter the study. This was done to ensure that information about the vulnerability was not leaked before the project owners could fix it. The study was conducted in virtual machines on the participants' working stations, and thus no risk to any live system was posed.

The study was reviewed and approved by our institutional ethics review board.

6.3 Methodology

To be able to compare the results from our freelancer sample with the student sample, we tried to replicate the study design of Plöger et al. as closely as possible.

In the following, we will present the similarities and point out and discuss the necessary differences between both studies.

6.3.1 Target Programs

Since the target programs and vulnerabilities are very influential in the study design, we used the same target programs, namely libroxml and tomlc99, in the same versions as in Plöger et al. To reduce the risk of participants circumventing the fuzzing task by looking for changes in newer versions of the program, we removed all information about the versions of the programs. We also thoroughly checked all submissions for hints of finding bugs by comparing the provided target programs with their current states. We were not able to find any.

libroxml

libroxml is an XML parsing library that consists of 13 source files with about 7800 lines of C code. For the configuration and compilation, it uses automake.

Bugs The bug in libroxml found by Plöger et al. and used in their study can be triggered by fuzzing the parsing function, arguably, the most exposed function of the library. Within the fuzz target, the participant only had to load an XML document. The fuzz target is straightforward, and no corpus, dictionary, or sanitizer needs to be used to trigger the bug.

tomlc99

tomlc99 is a parsing library for toml files written in C. It is comparably smaller than libroxml, with about 1200 lines of code. It has a handwritten Makefile for building.

Bugs The bug in tomlc99 found by Plöger et al. and used in their study can be found by parsing a toml file, but only if the address or memory sanitizer is used. During the study of Plöger et al., a participant found another bug in one of the executables of tomlc99. We were also able to confirm the bug. This bug can be found without using the memory or address sanitizer. However, a corpus is very helpful in finding the bug in a reasonable amount of time.

6.3.2 Fuzzers

To ensure the best possible comparison, we used the same fuzzers in the same version as in Plöger et al. Therefore, AFL in version 2.56b and clang [38] in version 10.0.1 was deployed. The current versions would have been 2.57b for AFL and 11.0.1 for clang. To the best of our ability to judge, none of the changes had anything to do with usability and thus plays no role in this study.

6.3.3 Study Design

We chose the same within-subjects study design as Plöger et al., as the within-subjects approach is best suited to account for individual differences in skill and has more power than a between-subjects study. We also randomized and counterbalanced the assignment of the fuzzers and the target programs to ensure that both were equally distributed over the participants' first and second tasks.

The participants started by filling out a pre-questionnaire where they were asked to rate and describe their programming skills as well as their demographic background. The pre-questionnaire was very similar to the one of Plöger et al. and can be found in Appendix C.1. After the pre-questionnaire, the two tasks were handed out consecutively. Participants had to keep a diary of their work on both tasks. Due to the much larger sample size than in the study by Plöger et al., we did not conduct interviews with all participants at the end of the study. Instead, we had all participants fill out a post-questionnaire, and only a subset was invited to a semi-structured interview, as Plöger et al. did. The communication with the freelancers was conducted via the freelancer.com messaging system as it is typical and mandatory for projects on the platform. This is in contrast to the study by Plöger et al. in which communication was done via email. We used the same playbook and phrasing used by Plöger et al. for all communication to ensure consistency in dealing with support requests.

The virtual machines supplied to the freelancers were also identical to that used by Plöger et al., with the exception that the Ubuntu OS was updated from 18.04 to 20.04, the latest stable version at that time. The update was done to prevent wasted time or other issues if a participant tried to update the system. Consequently, the GCC [74] compiler was updated from version 10.1.0 to version 10.3.0. With the update, no changes in the interaction of a participant with the system with respect to the fuzzing task were introduced. We do not assume that the update could have an impact on the results.

Recruitment

Since the main sign seen in the study by Plöger et al. was that the usability of AFL was slightly better than that of libFuzzer, we wanted to ensure that we would have enough statistical power to show statistical significance. So, we conducted an a priori power analysis for the pair-wise test of difference in the fuzzing score between AFL and libFuzzer based on the difference in means and standard deviation taken from Plöger et al. ($MD = 1.59$, $sd = 1.77$) on a scale from zero to ten. Since Plöger's study had a fairly small sample size, which increases the risk that measured differences in sample mean are larger than in the population [63], we set our target at a difference of mean of half that, i.e., $MD = 0.795$. We also think that any effect size smaller than that carries little practical relevance, so this is a good cutoff.

We calculated the necessary sample using the typical values for $\alpha = 0.05$ and $\beta = 0.2$. Using gPower [21], we calculated that we would need 61 participants to finish the study. We anticipated a completion rate between the completion rate of Plöger et al. of 36% of CS students and the one of Naiakshina et al. [136]

in their study with freelancers, which was about 90%. So initially, we aimed to recruit 100 freelancers.

On freelancer.com, one can hire two types of freelancers, preferred or non-preferred. A freelancer can get the status of being a preferred freelancer via an application. Minimum entry requirements include, besides other things being "top 3% overall ranking in one or more of your skills" and "active on the platform in the past 90 days". We hired a technical Co-pilot of freelancer.com who managed the recruitment process.

At first, we decided to only recruit from the pool of preferred freelancers with the same recruitment criteria as Plöger et al., namely proficiency in C/C++ and Linux. Moreover, we added the requirement of having sufficient communication skills in English since Naiakshina et al. [136] reported language difficulties with some freelancers.

As a part of the recruitment, freelancers were explicitly told that they would be taking part in a study. The recruitment included a very brief and simple check for C programming skills to ensure the freelancers could work with C-code.

We started with a batch of 10 participants.

However, only two of the first 10 participants finished the first task. Of the eight dropouts, two were classified as low-effort participants by our co-pilot since they claimed to have spent the entire time familiarizing themselves with the task. Since this also surprised our freelancer.com co-pilot, we interviewed the participants to understand if they understood the task and why they dropped out.

It turned out that the participants fully understood the task but found it very difficult to complete the task. They also reported that, as some of them were mostly working with C++ on a regular basis, they were not sufficiently familiar with C-code to be able to solve the task. Therefore, we adjusted the recruiting criteria from Plöger et al. to only include freelancers that were proficient in C instead of C or C++. Moreover, we removed the first subtask, which was "Get a first impression of the program libroxml and get an overall idea of what the program does.", to minimize the excuses for low-effort participants. We also increased our target sample start size from 100 to 200.

With the limitation of recruiting only preferred freelancers with proficiency in C and Linux, the pool of potential participants shrank to the point that our Co-Pilot contacted all freelancers from that category who are on the platform. Of those 284 preferred freelancers, 60 expressed interest in participating in the study. In the end, 37 of those decided to fill out the pre-questionnaire and started with the study. Thus, we decided to also recruit from the pool of non-preferred freelancers, which was significantly larger, containing 852 freelancers, and is in line with other studies using freelancer.com, such as those conducted by Naiakshina et al. and Danilova et al. From the pool of non-preferred freelancers, 177 expressed interest in taking part, and 113 started the study. In total, 237 freelancers expressed interest in participating in the study, 150 started and 62 finished. Among those who completed the study, 11 were preferred Freelancers, and 51 were non-preferred. Since our study uses a within-subjects design, all participants were exposed to all conditions. So while we did not hit our increased target of starting with 200 freelancers, the drop-out rate was not as large as suggested by our first batch, and the number of participants who completed the study is large enough to meet our power requirement.

Compensation / Payment

To determine the compensation, we took advice from the Technical Co-pilot of Freelancer.com, who recommended an hourly wage of 20€. With an approximate maximum workload of 21.5 hours, we paid freelancers 430€ for participating in the study. The total cost of the study, including the fees for the co-pilot, was just over 37,000€.

Time Frame and Limits

For each task, the participants had the same time limit of 10 hours to work on the task in a time frame of 10 days, as in Plöger et al. The time limit of 10 hours was strict, while the time frame could be extended on request. Since Plöger et al. always granted their students extensions to the time frame on request, we also started with this approach. However, in a few cases, freelancers asked for extensions multiple times, extending the time frame to several weeks or even months. Consequently, we decided to grant extensions of up to 30 days per task.

Task Structure

We utilized the same task description and task structure as Plöger et al. with one modification. As mentioned in Section 6.3.3 we omitted the first subtask *"Get a first impression of the target program and get an overall idea of what the program does"* after the first batch of 10 freelancers since we saw some claiming they had spent eight hours on that task and thus did not have time for the actually fuzzing subtasks. Thus, our participants only received the following six subtasks, which reflect the necessary steps of the fuzzing process:

1. *Find a suitable function to fuzz.*
2. *Write your fuzz target in an external file.*
3. *Compile and instrument the target program.*
4. *Compile the fuzz target.*
5. *Run the fuzzer and interpret the output.*
6. *If necessary, adjust and improve.*

Post-Questionnaire

Since it was foreseeable that such a large amount of participants could not all be interviewed, we decided to use a post-questionnaire to gather information on the participants' assessment of the fuzzing process, the fuzzers, and the comparison of both. The post-questionnaire can be found in Appendix C.2.

Semi-structured Interviews

While we opted not to interview all of our participants, we randomly selected participants from the batches for interviews since this would give us the capability to detect more subtle differences between the CS students and freelancers. After 11 interviews no new themes emerged. We then used purposeful recruiting to select another 7 participants, from which we hoped to get further and different perspectives on the usability of the fuzzers. In total, we conducted interviews with 18 of our participants.

For the semi-structured interviews, we used the same interview guideline as Plöger et al., which can be found in Appendix B.3.

6.3.4 Participants

The demographics of our participants who finished the study can be found in Table 6.1.

Gender	Male: 60	Female: 1	Other: 0	Prefer not to say: 1
Age	min: 16, max: 65	mean: 32.47, median: 31	sd = 10.33, NA = 0	

TABLE 6.1: Participants' Demographics

Looking at gender, the group of participants is very male-dominated, with only one female freelancer. This is an accurate representation of the freelancer population. The age of our participants is fairly widespread, between 16 and 65 years, with a median of 31 years.

We also asked our participants in which country they are currently living in. To give a better overview, we summarized their countries by continent from a geographical point of view. The summary can be found in Table 6.2. We associate a country that is part of multiple continents, e.g., Turkey or Russia, with one continent based on where most people of that country live. A complete listing of all nations participants are currently living in can be found in Appendix C.3.

Interestingly, compared to the freelancer study of Naiakshina et al. [136], our participants are far more distributed over the continents with a lesser focus on Asia. We still have a fairly large number of participants from India (10) and Pakistan (5), although, measured by the number of all participants, it does not seem unusual.

Continent	**n**
Africa	9
Asia	27
Europe	12
North-America	4
South-America	9
Oceania	0
NA	1

TABLE 6.2: Current country of our participants summarized by continent.

Programming and Fuzzing Proficiency

The programming experience of the participants that finished the study ranged from 1 to 42 years, with a mean of 10.60 years and a median of 7 years (sd = 8.80). This enables us to add the usability perspective of more experienced developers compared to the student sample of Plöger et al.

One participant reported a proficiency in C of 2, while all others reported a proficiency of at least 3. The median was 5 and the mean 5.23.

When asked if they were familiar with the term fuzzing 36 of the 62 participants answered the question with yes. Although, of those 36, only 11 had used a fuzzer before.

The usage of fuzzers was fairly spread, and no fuzzer was used by more than two participants before. The named fuzzers and tools can be grouped into web application fuzzing and non-web application fuzzing. For the non-web applications, fuzzers like libFuzzer, AFL/AFL++, honggfuzz, radamsa and OSS-Fuzz were named. wfuzz, dirb, dirbuster, ffuf, Monkey, and more were mentioned by our participants for web application fuzzing.

6.3.5 Support System

We used the same support system as Plöger et al., which was originally derived from Tiefenau et al. [181]. The support system was used to counter the problem of participants getting stuck early on and either delivering almost no data or dropping out entirely.

In the task description, the participants were told that if they had a problem that they could not solve on their own, they could contact the study assistant for advice after having invested a reasonable amount of effort. We stated that we would try to answer within one working day. We answered the first message of a topic within 24 hours between the working hours of the study assistant between nine a.m. and five p.m. and, as in Plöger et al., after a delay of 6 hours to not provide a quick way out with the support channel. As in Plöger et al., we gave support in the form of *hints* and *help*. For the first *hint*, we pointed the participants to the subtask that was causing the problem. In the second *hint*, we suggested taking a look at a specific part of the aforementioned subtask. After that, we gave *help* by providing a solution to their problem. We applied the same support system when a participant had submitted a solution but did not work on a subtask.

Any intervention from our side within the boundaries of the support system was counted as a failure for scoring the success of the affected subtask.

In contrast to the study of Plöger et al., as it seems to be common practice on this freelancing platform, we were asked by some participants to have short video calls to discuss questions and clarify the task, which we did. In the calls, the typical support behavior was maintained.

6.3.6 Scoring System

We used the same scoring system as Plöger et al. Every participant was able to get a fuzzing score between 0 and 10. We awarded a point for each of the following actions:

- Chose a useful function to fuzz
- Wrote a functional fuzz target
- Built the target program
- Instrumented all useful parts of the target program
- Built and instrumented the fuzz target
- Ran the fuzzer
- Used the address sanitizer
- Used a meaningful corpus or dictionary
- Triggered a bug
- Recognized a bug

As Plöger et al., we also accommodated alternative solutions in which participants combined or skipped steps in the fuzzing process. Therefore, we also awarded a point for each action if participants combined or skipped it if the participant reached the same end goal. This could be the case, for example, if a participant directly fuzzed the executable of the target program with AFL. In this case, they did not have to choose a suitable function to fuzz and write and build the fuzz target.

Successful Task Completion

We also use and report the measure of success of Plöger et al.

A participant completed their task successfully if they found and recognized a bug in the target program using the designated fuzzer without needing any support in doing so.

6.4 Limitations

In principle, this study inherits most of the limitations of Plöger et al.s study [157] This includes that only the fuzzers AFL and libFuzzer are evaluated; other fuzzers might produce different results. Furthermore, for comparability, we used the same two bugs as in Plöger et al. While Plöger et al. tried to find bugs of similar complexity, the match is not perfect. We counterbalanced the fuzzers and bugs to mitigate this issue, though.

The participants were guided with subtasks, and help was provided via the support system. While tutorials and forums also offer guidance and help and are used by many developers, we can only approximate the interaction and must be heedful to not be overly helpful. However, using this support system enabled us to gather far more detailed information on all fuzzing steps, compared to Plöger et al.'s first study where these systems were not in place [156]. In our view, the benefits far outweigh the drawbacks.

Last, but not least, all participants worked with code they were not familiar with. It is possible that their results would be different if they were fuzzing projects they had worked on before. Although as we will see in the following section, most of the problems we found seem fairly independent of the target programs and thus are likely to be attributable to usability issues of the fuzzers.

In addition, our study has the following differing limitations.

Sample

While Plöger et al. had the limitation of only having a CS student sample, our sample consists of freelancers from freelancer.com. As discussed in the related work section there are multiple different freelancer platforms which, while being similar, are not identical. Online freelancers also differ from developers employed by Western companies. Although the study done by Naiakshina [138] showed that the statistical significance and direction of effects were similar when comparing freelancers from freelancer.com and a sample of developers in German companies. Nonetheless, we point out that our sample is certainly not representative of all possible developers, and further research is needed for other subgroups.

Sample Size / Exploratory Study

The sample size in Plöger et al.'s study was very small and their analysis was focused on an exploratory approach with less emphasis on statistical testing. Since we used an a-priory power calculation to determine how many participants we would need to detect the main differences between AFL and libFuzzer reliably, we do not have this limitation.

Fixed bugs

Since we used the same target programs in the same version as Plöger et al., and they reported all known bugs to the developers, it is possible that participants used the online repositories of the target programs to get information on the bugs. We tried to mitigate this risk by removing all version information from the target programs. We analyzed the logbooks, bash history, exit surveys, and interviews for hints that this might have occurred and found no evidence of that.

Incentives

We believe the most relevant difference between the study of Plöger et al. and ours is participant type, i.e. CS students and freelancers. However, it is also possible that the different incentives had an effect. The students were rewarded with bonus points for their exam while the freelancers were paid their regular wages. It is possible that the students would have performed differently if they had been paid similarly. It is common practice to use different incentives for different types of participants, and, thus, any difference in results usually occurs in tandem anyway. More concretely, working with freelancers is significantly more expensive than working with students. The total cost of this study was just over 37,000€, and we are unaware of any study with CS students paying such high compensation. So from a methodological perspective, it is natural to compare outcomes, including typical incentives.

6.5 Results

In the following, we present the results of our freelancer fuzzing study.

When it is relevant, we compare the results of the preferred freelancer participants and the non-preferred freelancer participants. Overall, we found no major differences between the two groups.[1] Since the differences were small and not statistically significant by any means, we will report the majority of the results of all freelancers as one group.

[1]As far as we can tell the main difference is the payment/fee process on the freelancer platform and this has no effect on their performance.

6.5.1 Statistical Analyses

In this paper, we report the 95% confidence intervals (CI) instead of p-values since they more reliably communicate the uncertainty and scope of likely effect sizes, i.e., the true effect size is likely to lie within the CI. The larger the CI, the more uncertainty remains. If the CI of an odds ratio does not include the neutral 1, the result is considered statistically significant. Analog, if the CI of a mean difference does not include the neutral 0, the result is considered statistically significant. We leave α and β at the typical 0.05 and 0.8 values, respectively. All analysis was conducted using the statistical software R. The odds ratios for all between-subjects comparisons were calculated using the R-function `oddsratio.fisher`. For the odds ratios for the within-subjects tests, we used the R-function`cohenG` of the rcompanion package with 10,000 replications for the bootstrapping. We used the `MeanDiffCI` function, again from the rcompanion package, for the difference in means and their confidence intervals.

6.5.2 Fuzz-Target-Less Fuzzing, User Error Induced Crash and direct compilation

As in the study of Plöger et al., some participants used Fuzz-Target-Less (FTL) fuzzing when working with AFL. In this study, more than half of our participants, 36 out of 62, used FTL fuzzing, 19 in the first task and 17 in the second. These participants thus did not write a fuzz target but fuzzed the executable of the compilation process of the target program. This executable was directly fed to AFL. A very similar picture also emerged among CS Students. There, 9 of the 17 participants used FTL fuzzing.

If participants wrote fuzz targets it was possible for them to make mistakes that would lead to crashes that could be confused with the crashes fuzzers are supposed to find. Since the bug was contained in the participant's fuzz target it has no bearing on the target program. We denote these crashes in our analysis as UEIC (User Error Induced Crash).

Moreover, it was also possible to never build the target program as a whole with the configure and make commands but to only compile and link the needed parts of the target program and the fuzz target in one step.

6.5.3 Participants

Seven participants submitted solutions where they used the same fuzzer for both tasks instead of switching as instructed. All of them were asked to redo the second task with the designated fuzzer, which they agreed to do. As far as we could tell this did not impact their results.

Of the 150 participants who started the study, 64 participants finished the first task, and 62 of those finished the second task and the study. This gives us a completion rate of 41%, which is slightly higher than the rates of both previous fuzzing studies by Plöger et al. (28% and 36%). Nevertheless, the finishing rate is still lower than we would like, so we asked every participant who dropped out to share their reasons for their decision, to gain a better understanding and to hopefully be able to improve drop-out rates in the future.

We categorized the reasons of the 88 participants who dropped out or were excluded and summarized them. The corresponding Table can be found in Table 6.3. The reasons above the line are the drop-outs and below are the excluded.

Reason	**n**
No response	44
No expertise	21
No time	10
Personal Issues	4
Technical difficulties	2
Low effort	6
Outsourcing	1

TABLE 6.3: Reasons for dropping out of the study.

More than half of our participants did not respond to any of our communication attempts regarding their drop-out. Twenty-two participants stated they lacked the expertise to work on the task and thus wanted to drop out. The most common underlying reason for this was that participants perceived the complexity of the task to be very high, and they estimated that the effort to learn how to use the fuzzer would also be too high. In some cases, this was combined with a worry that poor performance would lead to a negative review on the platform. This is despite the fact that we made it clear in the study description that we were testing the fuzzers and not them, and there would be no negative outcome for them as long as they tried their best in the time given. All participants who completed the study got the same good review, and thanks for taking part in our study.

Initially, we had 9 low-effort participants. All of these were contacted via the co-pilot and 3 decided to improve their work and finish the study, but 6 decided to drop out.

Entertainingly, one enterprising participant tried to outsource the task by creating their own project on freelancer.com. The participant was removed from the study.

Finally, while conducting the study, we found that one participant had created a second account on the platform to take part in the study twice. Since participating multiple times was forbidden, we removed the second account from the study but let the first attempt stay in.

We monitored the platform and the results for further such attempts but could not find any others.

6.5.4 Completion Rate and Success

The overall statistics for all 150 participants regarding drop-outs, completion, and success rates can be seen in Table 6.4. The differences in drop-outs and completion between the two fuzzers were not statistically significant in our or in Plöger et al.'s study. The odds actually point in different directions. In our study, we found OR = 1.20, CI [0.67, 2.15] in favor of libFuzzer. Plöger et al found OR = 1.43, CI [0.51, 4.11] in favor of AFL. Since the results are not statistically significant, this is not a contradiction.In terms of failure and success when counting drop-outs as failures, both studies show a non-significant advantage to AFL: CS students (OR = 2.53, CI [0.68, 10.69]) and freelancers (OR = 2.06, CI [0.90, 5.00]).

The picture gets clearer when only looking at completed tasks of the 150 participants. Here we have a statistically and practically significant advantage for AFL: OR = 2.52, CI [1.03, 6.46]

Fuzzer	Order	started	drop-out	completed	success	
	1st	76	47	29	12	15%
AFL	2nd	35	1	34	10	29%
	∑	111	48	63	22	19%
	∑	103	40	63	11	11%
libFuzzer	2nd	29	1	28	5	17%
	1st	74	39	35	6	8%

TABLE 6.4: Overall drop-out, completion, and success of our participants for AFL and libFuzzer.

		libFuzzer		
		success	no success	∑
	success	5	17	22
AFL	no success	6	34	40
	∑	11	51	62

TABLE 6.5: Contingency table of outcome for AFL and libFuzzer in terms of success.

We now narrow down the analysis and only look at the 62 participants who completed both tasks and who finished the study. The outcome in terms of success and failure can be seen in Table 6.5. More than half of the participants (34) did not succeed with either of the fuzzers. In contrast to that, only 5 participants succeeded with both. For those participants who were able to solve the tasks with only one of the fuzzers successfully, AFL received better results than libFuzzer (17 vs 6). The odds ratio shows an advantage for AFL (OR = 2.83, CI [1.22, 10.00]) and is both statistically and practically significant.

Target Program Results

Target Program	started	drop-out	completed	success	
libroxml	107	44	63	19	19%
tomlc99	107	47	63	14	14%

TABLE 6.6: Overall drop-out, completion, and success of our participants for the target programs.

In the analysis of Plöger et al., there was uncertainty about whether libroxml was an easier target program. The results for our target program analysis can be seen in Table 6.6. Our results indicate a considerably smaller difference between the target programs (*OR* = 1.43, CI [0.64, 3.29]) than in Plöger et al. The difference is not statistically significant, and since we counterbalanced the fuzzers and programs, it should also not be practically relevant.

6.5.5 Fuzzing Score Results

This section contains the main comparison of the two fuzzers for which we conducted the power analysis. For our participants who finished the study, the mean and median fuzzing scores for AFL were 6.95 and 7, respectively. In comparison, the mean and median scores for libFuzzer were 5.63 and 6, resulting in a statistically significant difference in means of 1.32, CI [0.73, 1.91]. So, on average, the difference between a participant's score of AFL and libFuzzer is more than one point on a scale from 0 to 10.

The fuzzing scores of our freelancers are very similar to the results of the CS students in the study of Plöger et al (AFL: *M* = 8.24, *Mdn* = 9; libFuzzer *M* = 6.65, *Mdn* = 6). The scores of the freelancers are overall lower by about 1 point. However, the differences in means between the fuzzers are fairly similar, 1.32 for the freelancers to 1.59 for the CS students. Here we can see that the greater power of our study is helpful in making a clearer statement about the differences between the two fuzzers.

A detailed listing of the performance of all participants that finished the study in all the steps of the fuzzing process for the first and second tasks can be found in Table 6.7 and in Table 6.8. The tables have the identical structure and coding as those of Plöger et al.

When a participant received a *hint* in a step, it is denoted with an asterisk. For *help*, we used the plus sign. When a participant did not reach a step, it is denoted with a dash. Columns in gray are not part of the score.

As in Plöger et al., every step of a participant is color coded into four categories to provide further information about the performance. The categories are color-coded in the following way:

green An action is colored in green if the participant succeeded in this step without any support in this step or any dependent step before. This is equivalent to receiving a point for the score.

blue If a participant succeeded in a step without any support in it, but with support in a dependent step before, it is colored in blue.

yellow Yellow is used when a participant succeeded in the step but with support. Here we do not further distinguish between a first *hint*, a second *hint*, or *help*.

red If a participant did not succeed in a step, it is colored red.

For better comprehensibility, the structure and meanings of the columns, as described in Plöger et al., are repeated below.

Participant, Fuzzer and Program Those columns name the participant, the fuzzer that was assigned to the participant, and the target program for the task.

Sel. Func. The column Selected Function is two-folded. It shows how many functions the participant tried to fuzz, and the tick denotes if the bug could be triggered via one of those functions.

Working FT The number in the column Working Fuzz Target (FT) depicts the number of fuzz targets a participant wrote. The tick denotes whether a fuzz target could, in theory, trigger the bug.

Built TPr & Instr. TPr The column Built Target Program (TPr) contains the information on whether the target program was built, while the column Instrumented TargetProgram shows if the target program was instrumented.

Built FT The column Built Fuzz Target shows whether the fuzz target was built correctly.

Ran Fuzzer The column Ran Fuzzer shows if the fuzzer was run.

ASAN in TPr The column ASAN in Target Program shows whether the target program was built with the address sanitizer. The address sanitizer is of interest since it is needed to find the known bug in tomlc99. We awarded points in the libroxml condition as well since participants could not know that it was not needed, and it also helps when interpreting crashes.

Corpus/Dict The column Corpus/Dictionary shows whether participants used a corpus or dictionary. While this was not necessary to find the bugs, which the participants did not know, it is a common and recommendable optimization.

Trig. UEIC and Trig. Bug The column Triggered UEIC (User Error Induced Crash) shows whether a crash was induced by an error of the user and is not caused by an actual bug in the target program. A check mark means the participant did not trigger a UEIC, while a cross mark signals the opposite. The column Triggered Bug shows whether an actual bug in the target program was triggered.

Success The column Success shows if participants correctly interpreted the output as a bug, thus completing the overall task.

Score The fuzzing score is shown in the last column and is the sum of all subtasks which participants successfully solved without support.

6.5.6 Analysis of the fuzzing steps

If we look at the results from the perspective of the fuzzing steps, we better understand which steps were easy and which caused trouble for AFL and libFuzzer, respectively.

The corresponding data can be seen in Table 6.9. For each step and each fuzzer, a fraction and a percentage are shown. The denominator is the number of participants that reached that step. The numerator is the number of participants that successfully solved that step without needing support. The participants included in the numerator are those whose steps are marked in green and blue in Tables B.1 and B.2. The percentage is the percentage of the fraction.

Relevant differences between the fuzzers in favor of AFL can be seen in writing a correct fuzz target (AFL: 53/62 - LibFuzzer: 24/62), instrumenting the target program (49/62 - 30/62), and using a corpus or dictionary (56 - 38). Moreover, selecting a suitable function to fuzz seemed easier for our participants when using AFL. On the other side, libFuzzer had slight advantages when running the fuzzer and bigger advantages when using the address sanitizer. There were hardly any differences between the fuzzers for building the fuzz target and, interestingly, for correctly interpreting a bug.

Both similarities and differences to CS students can be seen in this comparison. Overall, the CS students had fewer problems in the fuzzing steps than the freelancers, which is also reflected in the higher overall fuzzing score. In the case of libFuzzer, writing a correct fuzz target and instrumenting the target program was a challenge for both groups. The benefit of the address sanitizer was higher for libFuzzer and is similar in both groups. Furthermore, building the target program was not a problem for the CS students or the freelancers.

Participant	Fuzzer	Program	Built TPr	Sel. Func.	Working FT	Instr. TPr	Built FT	Ran Fuzzer	ASAN in TPr	Corpus/Dict	Trig. UEIC	Trig. Bug	Success	Score
P015	AFL	libroxml	✓	✓ ./rocat (FTL fuzzing)**				✓	✗	✓	✓	✓	✗	1
P016	libFuzzer	libroxml	✓	✓	✗	✗*	✓	✓	✗*	✓	✗	✓	✓*	6
P019	AFL	libroxml	✓	✓ ./rocat (FTL fuzzing)+				✓+	✗	✗	✓	✓	✗	1
P021	AFL	tomlc99	✓	✓ ./toml_json (FTL fuzzing)				✓	✗	✓	✓	✓	✓	9
P022	libFuzzer	tomlc99	✓	✓	✓	✗*	✓	✓	✗	✗	✓	✗	-	5
P027	AFL	libroxml	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗*	8
P040	libFuzzer	libroxml	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✗	7
P045	AFL	tomlc99	✓	✓ ./toml_cat (FTL fuzzing)				✓	✗*	✗	✓	✗	✗	6
P047	AFL	tomlc99	✓	✓ ./toml_json (FTL fuzzing)				✓	✗	✓	✓	✓	✗*	8
P053	libFuzzer	tomlc99	✓	✗	✗	✗	-	-	✗	✗	-	-	-	1
P067	AFL	libroxml	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	9
P070	AFL	libroxml	✓**	✓	✓	✓ ./roxml (FTL fuzzing)		✓	✓	✓	✓	✓	✓	3
P071	libFuzzer	tomlc99	✓	✓	✗*	✗	✓	✓	✗	✓	✓	-	-	5
P072	libFuzzer	libroxml	✓	✓	✗	✗	✓	✓	✗	✗	✗	-	-	4
P077	libFuzzer	tomlc99	✓*	✓*	✓+	✓	✓*	✓	✓	✓	✗	✓	✓	1
P084	libFuzzer	tomlc99	✓	✓+	✓+	✓direct compilation		✓	✗	✗	✗+	✓	✓	4
P090	libFuzzer	tomlc99	✓	✓	✓+	✗	✓	✓	✗*	✗	✗	✓	✓	4
P092	libFuzzer	libroxml	✓	✓	✗+	✗	✓	✓	✗	✗	✗	-	-	4
P093	libFuzzer	libroxml	✓	✗**	✗	✗	✓	✓	✗	✗	✓	-	-	3
P094	AFL	tomlc99	✓	✓ ./toml_cat (FTL fuzzing)				✓	✗	✓	✓	✓	✓	9
P097	libFuzzer	libroxml	✓	✓	✓	✗	✓	✓	✗	✓	✗	✓	✓	8
P104	AFL	tomlc99	✓	✓ ./toml_cat (FTL fuzzing)				✓	✗	✓	✓	✓	✓	9
P108	libFuzzer	libroxml	✓	✓**	✓+	✓+	✓**	✓	✓	✓	✓	✓	✓	2
P112	AFL	libroxml	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	9
P113	AFL	libroxml	✓	✓ ./roxml (FTL fuzzing)				✓	✗	✓	✓	✓	✓*	8
P116	AFL	tomlc99	✓	✓ ./toml_cat (FTL fuzzing)				✓	✗	✓	✓	✗	-	7
P118	libFuzzer	libroxml	✓	✓	✗	✓direct compilation		✓	✓	✓	✗	-	-	7
P122	AFL	tomlc99	✓	✓ ./toml_cat (FTL fuzzing)				✓	✓*	✓	✓	✓	✓	7
P125	AFL	libroxml	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	9
P126	libFuzzer	tomlc99	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	9
P127	libFuzzer	tomlc99	✓	✓	✗	✗	✓	✓	✗	✗	✗	-	-	4
P128	AFL	tomlc99	✓	✓ ./toml_cat (FTL fuzzing)				✓	✗	✗	✓	✓	✓	8
P131	AFL	libroxml	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	9
P135	AFL	libroxml	✓	✓ ./roxml (FTL fuzzing)				✓**	✗	✓	✓	✓	✓**	6
P138	libFuzzer	tomlc99	✓	✓	✗	✗	✗	-	✗	✗	-	-	-	2
P139	libFuzzer	tomlc99	✓	✗	✗	✓direct compilation		✓	✓	✗	✗*	-	-	5
P144	libFuzzer	tomlc99	✓	✓	✓	✓direct compilation		✓	✓	✓	✗	✓	✓	10
P145	AFL	libroxml	✓	✓ ./roxml (FTL fuzzing)				✓	✗	✓	✓	✓	✓	9
P147	AFL	libroxml	✓	✓	✗	✓direct compilation		✓	✓	✓	✗	✓	✓*	8
P150	libFuzzer	libroxml	✓	✓	✗	✗	✓	✓	✗	✗	✗*	-	-	4
P154	AFL	libroxml	✓	✓ ./roxml (FTL fuzzing)				✓**	✗	✓	✓	✓	✓*	6
P157	libFuzzer	libroxml	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓	7
P159	AFL	tomlc99	✓	✓	✗	✓	✓	✓	✗	✓	✗	-	-	6
P160	libFuzzer	tomlc99	✓	✓	✗	✗	✓	✓	✗	✗	✗	-	-	4
P162	AFL	libroxml	✓	✓	✓	✗	✓	✓*	✗	✓	✓	✓	✗	5
P174	libFuzzer	tomlc99	✓	✓	✗	✓direct compilation		✓	✓	✓	✗	-	-	7
P177	AFL	tomlc99	✓	✓ ./toml_cat (FTL fuzzing)				✓	✓*	✓	✓	✓	✓*	7
P178	libFuzzer	tomlc99	✓	✗	✗+	✓	✓	✓	✗	✗	✗	-	-	2
P184	AFL	tomlc99	✓	✓ ./toml_cat (FTL fuzzing)				✓	✗	✓	✓	✓	✓	9
P190	libFuzzer	libroxml	✓	✓	✓	✓direct compilation		✓	✓	✓	✓	✓	✓	10
P193	AFL	libroxml	✓	✓	✓	✓direct compilation		✓	✗	✓	✓	✓	✓	9
P194	libFuzzer	libroxml	✓	✓*	✓	✗	✓	✓	✗	✓	✓	✗	-	5
P197	libFuzzer	tomlc99	✓	✓+	✓+	✗	✓+	✓	✗	✓	✗	✓	✗	2
P198	libFuzzer	tomlc99	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	9
P210	libFuzzer	tomlc99	✓	✓	✓	✓direct compilation		✓	✓	✓	✓	✓	✓*	9
P216	libFuzzer	libroxml	✓	✓	✗	✓direct compilation		✓	✗	✓	✗	-	-	6
P218	libFuzzer	tomlc99	✓	✓	✓*	✓direct compilation		✓	✓	✗	✗	✓	✓	6
P225	libFuzzer	tomlc99	✓	✓	✗	✓direct compilation		✓	✓✗	✗	✗	-	-	6
P226	AFL	libroxml	✓	✓ ./roxml (FTL fuzzing)				✓	✗	✓	✓	✓	✓	9
P228	AFL	tomlc99	✓	✓ ./toml_cat (FTL fuzzing)				✓	✓*	✓	✓	✓	✓*	7
P230	libFuzzer	libroxml	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	8
P235	libFuzzer	libroxml	✓	✓	✓**	✗	✓	✓	✗	✓	✓	✓	✗*	5

TABLE 6.7: Results of the 62 participants that finished the study for the first Task: ✓: success - ✗: failure - green: success without support - yellow: success with support - violet: success after support in previous step - red: no success - gray: not in score

On the other hand, when using AFL, the freelancers had more problems than the CS students with the instrumentation of the target program. Again for AFL, the correct interpretation of a crash as a bug was also harder for our freelancers. In addition, running the fuzzer also posed a challenge for a few of the freelancing developers which was not an issue for the CS students of Plöger et al.

6.5.7 User Rating of Fuzzers

As a part of the post-questionnaire, we asked our participants, to give both fuzzers an overall score and to rate their usability, both on a scale from 1, very bad, to 7, very good. The results of that can be seen in Table 6.10.

Participant	Fuzzer	Program	Built TPr	Sel. Func.	Working FT	Instr. TPr	Built FT	Ran Fuzzer	ASAN in TPr	Corpus/Dict	Trig. UEIC	Trig. Bug	Success	Score
P015	libFuzzer	tomlc99	✓	✗**	✗**	✗	✓**	✓	✗	✗	✗	-	-	1
P016	AFL	tomlc99	✓	✓	✓	✓*	✓	✓	✓**	✓	✗	✓	✓	6
P019	libFuzzer	tomlc99	✓	✓+	✗**	✓	✓**	✓	✗	✓	✓	-	-	1
P021	libFuzzer	libroxml	✓	✓	✗	✗	✓	✓	✗	✓	✗*	-	-	5
P022	AFL	libroxml	✓		✓ ./roxml (FTL fuzzing)			✓	✗	✓	✓	✓	✓	9
P027	libFuzzer	tomlc99	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	10
P040	AFL	tomlc99	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	9
P045	libFuzzer	libroxml	✓	✓	✗	✓direct compilation		✓	✗	✓	✗	-	-	6
P047	libFuzzer	libroxml	✓	✓	✓	✗	✓	✓	✗	✓	✗*	✓	✓	7
P053	AFL	libroxml	✓		✓ ./roxml (FTL fuzzing)			✓	✗	✓	✓	✓	✗*	8
P067	libFuzzer	tomlc99	✓	✓	✓	✓direct compilation		✓	✓	✓	✓	✓	✓	10
P070	libFuzzer	tomlc99	✓	✓+	✓*	✗	✓	✓	✗	✗	✗+	✗	-	3
P071	AFL	libroxml	✓	✓	✗+	✗	✓	✓	✗*	✓	✓	-	-	5
P072	AFL	tomlc99	✓		✓ ./toml_cat (FTL fuzzing)			✓	✗	✓	✗	✗	-	7
P077	AFL	libroxml	✓		✓ ./rocat (FTL fuzzing)**			✓	✗	✓	✗	✓	✓**	2
P084	AFL	libroxml	✓		✓ ./roxml (FTL fuzzing)			✓*	✗	✗	✗	✓	✓**	5
P090	AFL	libroxml	✓**	✓	✓	✓ ./roxml (FTL fuzzing)		✓*	✗*	✓	✗	✓	✓*	3
P092	AFL	tomlc99	✓	✓	✓+	✓direct compilation		✓	✓*	✓	✗	✓	✓	6
P093	AFL	tomlc99	✓		✓ ./toml_cat (FTL fuzzing)			✓	✓	✓	✓	✓	✓	10
P094	libFuzzer	libroxml	✓	✗	✗	✗	✓	✓	✗	✓	✗	✓	✓	6
P097	AFL	tomlc99	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	-	7
P104	libFuzzer	libroxml	✓	✓	✓*	✗	✓	✓	✗	✓	✗	✓	✗	5
P108	AFL	tomlc99	✓	✓+	✓	✓	✓	✓*	✓**	✓	✗	✓	✓*	5
P112	libFuzzer	tomlc99	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	10
P113	libFuzzer	tomlc99	✓	✓	✓	✓direct compilation		✓	✓**	✓	✗	✓	✓	7
P116	libFuzzer	libroxml	✓	✓	✗	✗	✓**	✓	✗	✗	✗	✓	✓	5
P118	AFL	tomlc99	✓	✓	✓	✗*	✓	✓	✗	✓	✗	✗	-	6
P122	libFuzzer	libroxml	✓	✓	✗	✗	✓	✓	✗	✓	✗	-	-	5
P125	libFuzzer	tomlc99	✓	✓	✓	✗*	✓	✓	✓	✓	✓	✓	✓	6
P126	AFL	libroxml	✓		✓ ./rocat (FTL fuzzing)			✓	✗	✓	✓	✓	✓*	8
P127	AFL	libroxml	✓		✓ ./roxml (FTL fuzzing)			✓	✗	✓	✓	✓	✓	9
P128	libFuzzer	libroxml	✓	✗	✗	✓	✓	✗	✓	✗	✓	-	-	4
P131	libFuzzer	tomlc99	✓	✓	✓	✓direct compilation		✓	✓	✓	✓	✓	✓	9
P135	libFuzzer	tomlc99	✓	✓	✗+	✓direct compilation		✓	✓**	✓	✗*	✓	✓	6
P138	AFL	libroxml	✓	✓	✓	✓ ./toml_cat (FTL fuzzing)		-	✗	✗	-	-	-	3
P139	AFL	libroxml	✓		✓ ./rocat (FTL fuzzing)			✓	✗	✓	✓	✓	✓	9
P144	AFL	libroxml	✓	✓	✓	✓direct compilation		✓	✗	✓	✓	✓	✓	9
P145	libFuzzer	tomlc99	✓	✓	✓	✓	✓	✓	✓*	✓	✓	✓	✓	7
P147	libFuzzer	tomlc99	✓	✓	✓+	✓direct compilation		✓	✓	✗	✗**	✓	✗*	6
P150	AFL	tomlc99	✓	✓	✓	✗**	✓	✓	✗	✓	✓	✗	-	6
P154	libFuzzer	tomlc99	✓	✓+	✗	✓	✓	✓	✓	✗	✗**	-	-	5
P157	AFL	tomlc99	✓		✓ ./toml_cat (FTL fuzzing)			✓	✓*	✓	✓	✓	✓	7
P159	libFuzzer	libroxml	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	10
P160	AFL	libroxml	✓	✓	✓	✓	✓	✓	✗	✓	✗*	✓	✓*	7
P162	libFuzzer	tomlc99	✓	✓	✓+	✓direct compilation		✓	✓*	✓	✗	✓	✓*	6
P174	AFL	libroxml	✓		✓ ./roxml (FTL fuzzing)			✗	✗	✓	✓	✗	-	6
P177	libFuzzer	libroxml	✓	✓	✗+	✓direct compilation		✓	✗	✓	✓	-	-	6
P178	AFL	libroxml	✓	✓	✓	✓ ./roxml (FTL fuzzing)*		✓**	✗	✓	✗	✓	✗	4
P184	libFuzzer	libroxml	✓	✓	✓	✗	✓	✓	✗	✓	✗	✓	✓	7
P190	AFL	tomlc99	✓		✓ ./toml_cat (FTL fuzzing)			✓	✗	✓	✓	✓	✓	9
P193	libFuzzer	tomlc99	✓	✓	✓	✓direct compilation		✓	✓	✓	✓	✓	✗	9
P194	AFL	tomlc99	✓		✓ ./toml_cat (FTL fuzzing)			✓	✓*	✓	✓	✓	✓	7
P197	AFL	libroxml	✓	✓*	✓*	✓+	✓	✓	✓	✗	✗	✓	✗*	2
P198	AFL	libroxml	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	8
P210	AFL	libroxml	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	5
P216	AFL	tomlc99	✓	✓	✓	✓direct compilation		✓	✓	✓	✓	✓	✓	10
P218	AFL	libroxml	✓		✓ ./rocat (FTL fuzzing)			✓	✗	✓	✓	✓	✓	9
P225	AFL	libroxml	✓	✗	✗+	✓direct compilation		✓	✓	✓	✓	-	-	6
P226	libFuzzer	tomlc99	✓	✗	✓	✗	✓	✗	✗	✓	-	-	-	4
P228	libFuzzer	libroxml	✓	✓	✓	✗	✓**	✓	✗	✓	✓	✓	✗	4
P230	AFL	tomlc99	✓	✓	✓	✓	✓	✓	✓*	✓	✓	✓	✓	7
P235	AFL	tomlc99	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	10

TABLE 6.8: Results of the 62 participants that finished the study for the second Task: ✓: success - ✗: failure - green: success without support - yellow: success with support - violet: success after support in previous step - red: no success - gray: not in score

The mean and median usability score given by our participants for AFL was 5.59 and 6, respectively. For libFuzzer, the scores were 4.92 and 5. Therefore, the difference in means in the usability score of both fuzzers is 0.67 in favor of AFL, CI [0.24, 1.09] . The difference of two-thirds of a point on a 7-point scale can be noticeable from a practical point of view. However, the lower bound of the CI is only 0.24 which would not be a practically relevant difference. Thus while we can conclude that our participants rate the usability of AFL better, the lead might be quite small.

However, this is still a clear contrast to the CS students, where the usability score of both fuzzers was nearly dead even, and no difference could be seen (DM = 0.06, CI [-1.21, 1.32]). AFL performed noticeably better for the usability

Fuzzing Steps	AFL		libFuzzer	
Built target program	60/62	97%	60/62	97%
Selected function	56/62	90%	46/62	74%
Wrote correct fuzz target	53/62	85%	24/62	39%
Instrumented target program	49/62	79%	30/62	48%
Built fuzz target	57/62	92%	53/61	87%
Ran fuzzer	51/61	84%	58/60	97%
Used ASAN	8/62	13%	22/62	35%
Used Corpus/Dict	56/62	90%	38/62	61%
Correctly interpreted bug	30/51	59%	22/35	63%

TABLE 6.9: Ratio and success percentage of steps by fuzzer.

Overall										Usability									
Fuzzer	1	2	3	4	5	6	7	plot	median	Fuzzer	1	2	3	4	5	6	7	plot	median
AFL	0	0	5	3	16	28	11		6	AFL	0	2	4	11	15	15	16		5
libFuzzer	3	0	7	14	12	19	9		5	libFuzzer	3	5	9	13	13	14	6		5

TABLE 6.10: Usability and Overall rating of AFL and libFuzzer given by the participants

score for the freelancers, while libFuzzer performed similarly for both groups.

The overall score is quite similar to the usability score for the freelancers. The mean and median for AFL are 5.35 and 5, and for libFuzzer, the scores are 4.49 and 5. Even though the median of both is the same, the difference in means with 0.86 (CI [0.39, 1.33]) is even higher in comparison to the usability score. Likewise, the lower bound of the confidence interval is higher so that the difference in any plausible cases is noticeable.

For the overall score, the results for the freelancers are very similar to those of the CS students (DM = 0.88, CI [-0.04, 1.81]). Both prefer AFL overall, and the difference between the two fuzzers is nearly identical.

Moreover, we also asked our participants, if they wanted to use one of the fuzzers in the future. The results are shown in Figure 6.1. More than half of our participants reported that they wanted to use both fuzzers in the future, while only one participant did not want to use either. Of those participants who only picked one fuzzer AFL was far more popular. The odds ratio is 3.14 in favor of AFL (CI [1.45, 9.67]), showing a strong tendency.

Again, CS students (OR = 4.5, CI [1.2, 12]) and freelancers were of the same opinion about the usage of a fuzzer in the future.

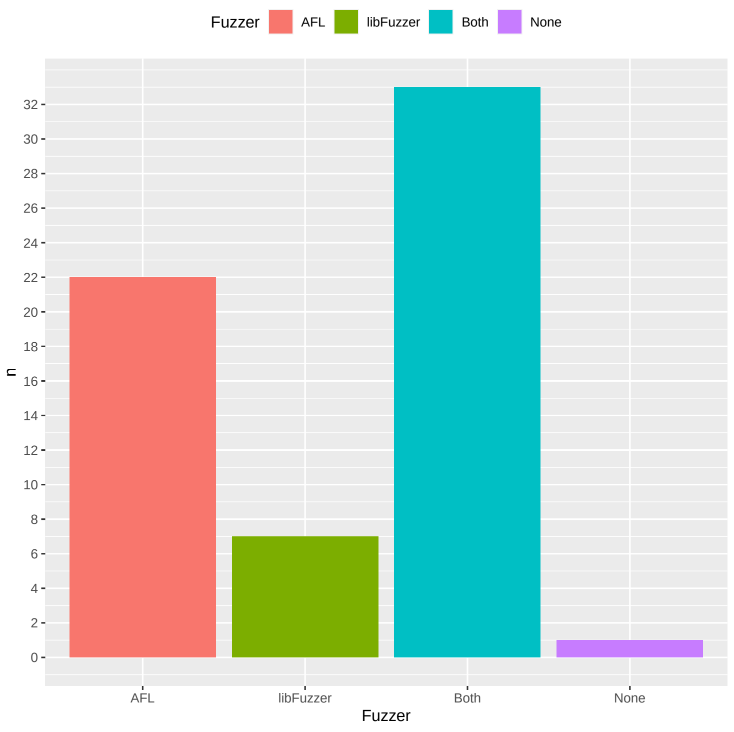


FIGURE 6.1: Number of participants that intended to use AFL or libFuzzer in the future.

6.5.8 Support System Insights

Over all 150 Participants, we gave 194 *hints*, 92 in the first task and 102 in the second task. We also *helped* 19 times in the first task and 15 times in the second task. In the first task, we gave *hints* to 41 participants, of whom 8 dropped out in that task. So, more than half of all participants that finished the first task received a *hint*. For the second task, 38 participants (59%), received a *hint*, and none dropped out. Overall, *hints* were given to 56 different participants.

An overview of *hints* and *help* is given for both tasks for every step of the fuzzing process, and both fuzzers can be seen in Figure 6.2. Besides the steps, we also added information on the support given in the situation where FTL fuzzing was used.

The analysis of crashes was also a step where participants often needed support. In principle, it included checking whether a crash was a security-related bug or not. There were two possible scenarios. On the one hand, the crash could stem from a bug in the target program, and the participant had to analyze it in order to find out that the crash was indeed a bug. On the other hand, the crash could stem from a UEIC. In this case, the participant had to understand that they made a mistake and to correct their fuzz target.

For AFL, especially in comparison to libFuzzer, a lot of *hints* were given for actually starting the fuzzer and working with the address sanitizer. In contrast

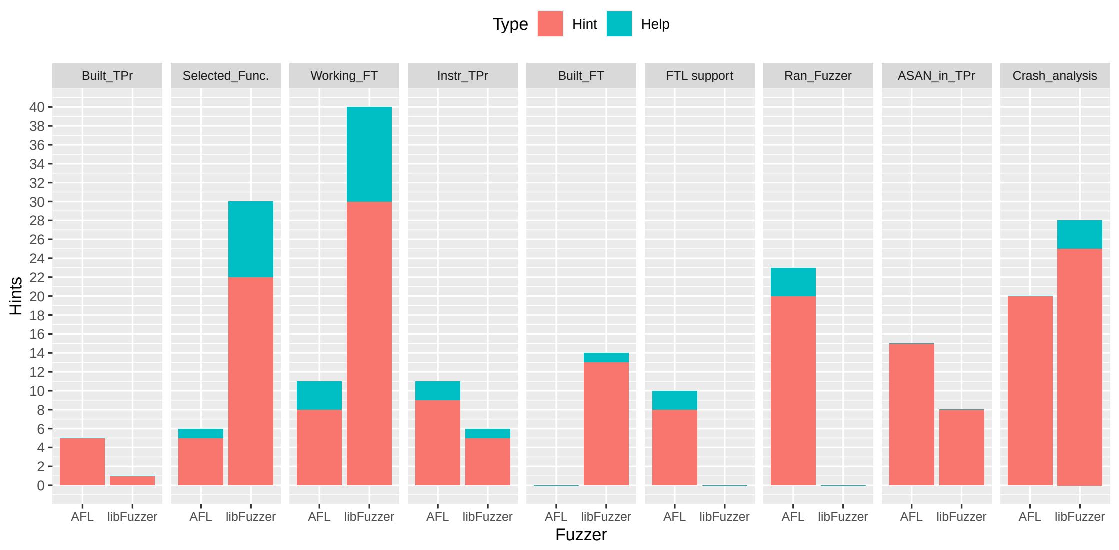


FIGURE 6.2: Hints and Help given per steps for both fuzzers

to that, a large amount of *hints* and *help* was given for selecting a suitable function to fuzz and writing a working fuzz target in the case of libFuzzer. For both fuzzers support was needed for the analysis of crashes.

Large differences can be seen when we compare the graphs of support given for the freelancers and the CS students. We attribute this mostly to the fact the freelance developers struggled more and earlier in the fuzzing process. Therefore, the amount of support for selecting a suitable function to fuzz and writing a working fuzz target is comparably high. Those numbers are for the freelancers lower in the case of AFL, most likely because many of them used FTL fuzzing and because a fuzz target writing for AFL automatically provides a null-terminated string causing the user to not run into out-of-bounds access problems.

Support during the instrumentation of the target program and the usage of the address sanitizer is comparably low, potentially because there was less time left to give support in these steps. But the failure rate was similar between the groups. We will get into more detail about those findings below.

6.6 Analysis of Fuzzing Steps

For the analysis of fuzzing steps, we analyzed the participant's working directories, the post-questionnaire as well as the diaries and double-checked our findings with the interviews.

6.6.1 Selected Function

For the selection of a suitable function to fuzz, our participants can be divided into two groups. The first group, which is the majority of the participants, namely 43, choose to use a function to read in a toml or XML file for the reason that they thought it was obvious and well-suited for good coverage:

> it was the obvious choice (P097)

> [the] function provides great coverage as it is the root node in this Library. (P174)

The second group, however, tried to find a function that was the easiest to use, especially in combination with the provided arguments of libFuzzer. Often STRNDUP was chosen for this reason. The function duplicates a string by taking a string in the form of a char pointer as input, allocating a buffer of the same size, copying the content of the input into the buffer, and returning a pointer to that buffer. Even though the function STRNDUP was very small (9 lines of code,) participants often made mistakes leading to two UEICs when fuzzing it. Firstly, they did not provide a string, but only a char pointer as an input which resulted in a direct crash if the address sanitizer was used, as the size of the input could not be determined, causing a read outside of the buffer. Secondly, the created buffer was not freed in the fuzz target after receiving the pointer to it, causing an out-of-memory error.

This shows a certain difference to the 17 CS students, as this step seemed to cause more problems for the freelancers.

6.6.2 Working FT

Writing a working fuzz target was arguably the most complex task for our freelancers, especially when using libFuzzer. In the case of libFuzzer, using the provided arguments of the LLVMFuzzerTestOneInput function, Data and Size posed the by far greatest challenge. They were either not used at all, at the wrong place or, when a string was needed, not null-terminated.

AFL, on the other hand, was easier to use for our participants in that case. A not to be underestimated number of participants, namely 36, used FTL fuzzing which circumvents the problem of writing a fuzz target. Moreover, working with a main function seemed to be the more intuitive way for our participants as the LLVMFuzzerTestOneInput function overall led to confusion. Furthermore, the input received from the main function or when using a read-in function is automatically null-terminated, which prevented a for libFuzzer typical out-of-bounds access.

These problems affected both the freelancers and the CS students. Both struggled to create working fuzz targets.

6.6.3 Building and Instrumenting the Target Program

The instrumentation of the target program was also a major In Table 6.9 it can be seen that libFuzzer performs a lot worse than AFL. This is due in large part to AFL FTL mode. If we exclude the 33 times that participants used FTL fuzzing, the success rate in this step for AFL shrinks to 55%, which is comparable to the 47% of libFuzzer.

There were several reasons for the difficulties. Participants both overlooked the need for instrumentation but also had technical difficulties with the process with many hoping that it would work without it.

This is similar to the CS students, where the problems resulted in a guessing game to find the right commands to make it work. Oftentimes, it ended in using *make install* to install the library system-wide to make it easier to link against it. Typically, this was accompanied by the fact that the target program was not instrumented.

6.6.4 FTL Support

In contrast to the CS students of Plöger et al., a few of the freelancers needed support when using FTL fuzzing. This included either compiling the target program in a way that the executable is instrumented or helping the participant to find the right executable after the compilation and instrumentation.

6.6.5 Ran Fuzzer

For the freelancers, there was no difference compared to the CS students of Plöger et al. when it came to running the fuzzer. For libFuzzer, no problems were present. However, AFL had the same problem with starting the fuzzer. Our participants struggled with the "@@" option of AFL. Similar to the CS students, they did not understand that in both cases, the path to the corpus is provided but that in the case without the "@@" it is provided via standard in while with the "@@" it is provided as a command line argument. This resulted in the situation that the fuzzer showed a red warning in the main window stating "(Last new path : none yet (odd, check syntax!))". In contrast to the CS students, who were able to overcome the problem by themselves, 7 freelancers overlooked the warning, ran the fuzzer from a couple of minutes to hours, and concluded that no bug existed in the target program. If they concluded this and handed in their work we used the support channel to request that they analyze everything marked in red in the output, causing them to discover and work on the problem.

Even though AFL provides feedback in the form of telling the user that no paths are discovered, it would be more useful if more specific feedback could be provided. Our suggestion would be that AFL automatically starts with and without the "@@" option to see which produces better coverage. Even one second with each option would most likely tell a clear story of which option is the better one, which would improve usability.

6.6.6 Address Sanitizer

While none of the freelancers struggled using the address sanitizer or any other sanitizer, only a handful used them of their own volition. Overall, 35% of the participants used the address sanitizer when using libFuzzer, while only 13% used it in conjunction with AFL. This is similar to the findings of Plöger et al. for the CS students.

6.6.7 Corpus/Dictionary

We do not consider it surprising that the vast majority of our participants used a meaningful corpus or dictionary while working with AFL, as it is a necessity to provide one to start the fuzzer. The majority used a corpus, a few used both, none used only a dictionary With libFuzzer we see that only 61% used a corpus, which we attribute to the fact that it is not mandatory. Interestingly, the use of a corpus is more common among freelancers than among students.

6.6.8 Fuzzing-run Output

In the post-questionnaire, 43 participants reported that they studied the output of both fuzzers while they were running. Some of the participants couldn't because they did not reach that step and for some a crash occurred immediately, so they could not evaluate the runtime output. We asked the 43 participants who saw the output of the fuzzers while running on a scale from 1, very bad, to 7, very good. The mean and median of AFL were 5.91 and 6, and for libFuzzer, 4.72 and 5. The difference in means was 1.19, CI [0.67, 1.70] and shows a significant preference for AFLs output. This data was not gathered from the CS students.

In terms of deciding whether the fuzzer was running properly, the freelancers behaved very similarly to the CS students. They checked whether the fuzzer was finding new paths or if new unique crashes were being found.

6.6.9 Crash Output

We also asked the participants to rate the output when a crash occurred on a scale from 1, very bad, to 7, very good. This question was answered by 42 participants. The mean and median of AFL were 5.33 and 6, and for libFuzzer, 4.76 and 5. The difference in means was 0.57, CI [-0.16, 1.30]. This difference is smaller than for the output while running and is not statistically significant. However, it needs to be taken into account that, as we found out in the interviews, this grade is mostly based on the direct feedback in the interface of AFL as well as the structure and form of the output in terms of crash files and their locations. It is less about getting further information about the crash by the fuzzer.

Therefore, the situation seems to be more in line with the statements of the CS students that the notification about crashes and the cleanness of the structure of AFL was good, but the information provided is insufficient. On the other hand, more information on the actual crash was provided by libFuzzer, especially in the case that the address sanitizer was used. This highlights an interesting divergence between usability and functionality.

6.7 Discussion

While our completion rate of 41% is higher than in the previous two studies by Plöger et al., we were still surprised by the high drop out rate. Since we were paying the freelancers regular wages and they were not being judged on results but merely for taking part, we had expected higher completion rates. Unlike Plöger et al., who attribute this to both the complexity and length of the task, we think for our sample, it is mainly the complexity. The fact that the most named reason for dropping out of the study was not having enough expertise to work on the task emphasizes this.

This shows that the poor results found by Plöger et al. with the CS students replicated very clearly with the freelancers. Consequently, we can now be more confident that it is worthwhile investing more effort into improving the usability of these powerful tools since it is not only inexperienced students who have trouble.

In the following, we will go into more detail with a quantitative and qualitative comparison of the CS students and the freelancers.

6.7.1 Quantitative Analysis

From a quantitative point of view, the CS students and freelance developers performed very similarly in relative terms. For success as well as for the fuzzing score, both groups performed better with AFL than with libFuzzer.

From an absolute point of view, the CS students interestingly performed better with both fuzzers.

Small differences occur for the proportion of dropouts and completions with the respective fuzzers. More CS participants dropped out while using libFuzzer in contrast to the freelancers, where more participants dropped out using AFL. However, the effect sizes are quite small and were not statistically significant. Another observation is that a noticeable difference in terms of the proportion of failure and success with respect to the target programs could be found for the CS students. This effect is much lower for the freelance developers.

Also, the self-reported opinions are very similar. The overall score, as well as the intended usage in the future, show the same preference for AFL. Only the usability score, which was a surprise in the study with the CS students, differs. While the usability score for the freelancers shows a strong tendency towards AFL, the CS students did not have a preference between the two.

6.7.2 Qualitative Analysis

The qualitative analysis also reveals many similarities between the two groups. Writing a meaningful fuzz target was a complex task for both groups. Therefore, using FTL fuzzing was a more user-friendly alternative that many participants chose. Moreover, building and instrumenting all necessary code was, for many participants, a guessing game, where *make install* was a method of last resort. Also, the output while running the fuzzer and the output presented when a crash occurred were perceived similarly.

Differences could be seen, especially in choosing a suitable function to fuzz, where only very few CS students had trouble doing so and where a noticeable number of freelancers struggled. This is also reflected in the support statistic. The step of running the fuzzer is another example where we can see a slightly better overall performance of the CS students. In both groups, participants encountered the problem of starting AFL the right way. However, nearly all of the CS students overcame the problem, while a lot of freelancers needed support to do so.

6.7.3 Recommendations

Based on the increased statistical power and different population sample of our study, we want to strengthen the recommendations made by Plöger et al. These will benefit not only inexperienced CS students but the wider developer community as well.

6.8 Conclusion

In this paper, we replicated Plöger et al.'s small-scale CS student fuzzing study with a larger sample of freelance developers. We conducted an a priori power analysis to make sure our sample was large enough for our statistical analysis. Of the 150 freelancers we hired, 62 completed the full 20-hour study with both AFL and libFuzzer. We were able to replicate Plöger et al.'s results finding the same relative results in all major comparisons between the fuzzers. Although interestingly, the freelancers performed slightly worse in absolute terms. Our results show that the usability of fuzzers still needs significant improvements - even more so for the freelance developers than for CS students. We think it is a worthwhile goal to improve usability for non-expert developers since there are not enough fuzzing experts to go around, and it is unrealistic that we train all developers to become fuzzing experts. So enabling these kinds of developers to use fuzzing effortlessly would be a sea change and enable powerful software testing at scale.

Chapter 7

CS Students Fuzzing Their Own Code or Code Written by Others

Disclaimer: *This chapter's content was joined work with my co-authors Mischa Meier, Christian Tiefenau, and Matthew Smith. As this work was conducted with my co-authors as a team, this chapter will use the academic "we" to mirror this fact. I designed the study with advice from Matthew Smith. I set up and performed the study as well as designed and conducted all interviews. Mischa Meier supported me in the coding and the discussion of the interviews. He, Christian Tiefenau, and I carried out the scoring, and we discussed the results.*

Finally, before compiling the paper for publication, Matthew Smith and I jointly discussed the study's implications.

7.1 Motivation

Fuzzing is one of the hot topics in academic research, with more than 200 published papers about it since 2020 [28]. The popularity of fuzzing is also rapidly rising in industry. Large software companies such as Microsoft[127], CISCO [37], Deutsche Telekom [56] and Google [40] are using fuzzing extensively to make their software more secure.

While it would be ideal if all companies that produce code also hire internal or external security testing experts, many organizations do not have the means to do so. It is still recommendable for organizations to implement a code security testing process in which their software engineers/developers are involved in testing. A question that arises in this context is whether it would be better if developers test their own code because they are familiar with it or whether it would be better if a peer-based testing strategy is implemented where developers test the code of team members. The idea is that a fresh set of eyes - potentially also more critical eyes - serve better than having developers test their own code. To the best of our knowledge, there is no empirical evidence to answer this question.

As a first step in shedding light on this question, we present the first randomized control trial with code-authorship being the independent variable to gather empirical evidence on the effectiveness of fuzzing one's own code vs. fuzzing code written by a team member. We conducted our study with 35 CS students who were fuzzing novices and had taken part in a system-level programming module in which they learned C and assembly and took part in programming a shell over the course of a semester. We tasked the students either with fuzzing their own code or code from another participant with libFuzzer. We also extend insights into the usability of libFuzzer for novice users. Specifically, we extend the insights from Plöger et al. [156, 157] on usability issues when using libFuzzer to find bugs and present the first usability insights into using libFuzzer's output to fix the bugs found. We argue that this novice perspective is essential since, as we argued, there are not enough fuzzing experts to go around. Hence it would be beneficial if powerful testing tools like fuzzing were so easy to use that even novices without any security specialization could benefit from them.

Table 7.1 gives an overview of the different types of testers. In this paper, we focus on the last two rows, i.e., the internal testers without specialized security/fuzzing knowledge, since, in our experience, these are the most common resources organizations can deploy, so helping them would be of great benefit.

Tester	Organisation	Testing Specialization	Project Familiarity	Wrote the Code
External specialist	external	yes	no	no
Internal specialist	internal	yes	yes	no
Peer tester	internal	no	yes	no
Self testing	internal	no	yes	yes

TABLE 7.1: Overview of different types of testers

7.2 Methodology

In order to examine the effect of fuzzing one's own code compared to fuzzing code written by others, we needed a study setup fulfilling several requirements:

1) we needed enough participants who have written code that can be tested to have a decent chance of finding statistically significant effects if they are present in the population.

2) The codebase of the participants needs to be large enough to not just be a toy example and it needs to be of similar size and complexity to the code of the other participants so that the code does not create a confounding effect.

3) We need to find an equal number of participants who are similar enough to 1 that they can be used as the "other" group without confounding the results.

Due to these requirements recruiting company developers or developers from open-source projects was not an option. Firstly, it is unlikely that one would get enough. Secondly, both they and their code are likely heterogeneous and thirdly, it would be very hard to find enough similar developers for the "other" group. Instead, like Plöger et al., we recruited CS students - in our case, however, from a module in system-level programming. We picked this module because, over the course of the semester, students incrementally built a shell in C and assembly from scratch, i.e., students were prohibited from using any external library such as the C standard library. This gave us an excellent environment to conduct our study since we had a large number of fairly similar participants who were all familiar with the same moderately complex code base, which we could extend for our study. This allowed us to isolate the effect of fuzzing one's own code better than with any other participants or code we could think of.

While the use of CS students from this course allowed us to fulfill requirements 1-3, it does have the limitation that they are not the same as seasoned developers, and thus, this limits generalisability. However, Naiakshina et al. [135] found that students can be a helpful substitute for professionals in a programming study, especially since this is an exploratory study. We think the benefits of using CS students outweigh the downsides.

Thus, we recruited students from this module and had them fill out a short pre-questionnaire (10 minutes). Then we assigned them to a four-hour programming task that extended the shell project. Since all students had built the shell, we needed to create code for the study to be fuzzed that was not known to all participants. So we randomly assigned half the participants to a parser programming task (treatment task/TT) or a four-hour color converter programming task (control task/CT).

After that, all participants took part in a ten-hour fuzzing task. Those participants who were assigned the programming treatment task (TT) fuzzed their own code (own-group). Those in the control condition were randomly assigned to fuzz the code from a participant of the own-group and thus fuzzed code written by someone else (other-group). The ten hours could be spread freely over the course of ten days. These times were taken from the fuzzing studies by Plöger et al.

During the two tasks, participants were asked to keep a detailed diary about their work and how much time they invested in the different steps.

Once they completed the fuzzing task, participants were asked to fill out a post-questionnaire (20 minutes) and had to participate in a final interview (30 minutes).

In the following, we will discuss the design decisions in more detail.

7.2.1 Shell & Programming Tasks

We designed programming tasks to extend the shell for several reasons. Firstly, if we had just used a stand-alone programming task, the code would either have been very simple and thus unrealistic for the other-group or the programming task would have had to be prohibitively long. Thus, adding the shell gave us the benefit of having a more realistic complexity within a feasible time frame. But since all students recruited for the study had programmed and discussed parts of the shell, we deemed those components too familiar even if they should fuzz someone else's implementation. Thus, we decided to add a treatment task, the code later to be fuzzed, and a control task. The control task was needed to make sure that all participants had four hours of programming before starting to fuzz.

Treatment-Task

For the treatment task, we needed a programming task that could be completed in a short amount of time, that was fairly likely to contain bugs and was simple to fuzz. So we chose to task participants with implementing a parser to read a configuration file for the shell. We chose parsing because a simple parser can be coded in a couple of hours, the input, a string, is comparably easy to create out of the Data and Size argument of the LLVMFuzzerTestOneInput function of libFuzzer and correctly implementing parsing has proven to be error-prone. Just as with the shell, participants were not allowed to use libraries and had to implement everything from scratch.

To keep the task as short as possible, we provided an example configuration file, the code to read the configuration file from disk, and the header of the parsing function. So the participants only had to implement the actual parsing functionality. The parsing task was to set a Welcome- and End-Message for the shell. Only lines with the structure: *Keyword:Value* should be considered, with *Keyword* having two possible values namely *WelcomeMessage* and *EndMessage*. The value of *Value* can be any string.

Our research hypothesis is that taking part in the treatment task has an effect on the fuzzing outcome. This could either be a positive effect, e.g., because being familiar with the code being fuzzed makes it easier to write a good fuzz-target, or a negative effect, e.g., because familiarity leads to things being overlooked. Consequently, all statistical testing is done two-tailed.

Control-Task

In the Control-Task, participants had to extend the shell by implementing a color converter as a helper function to change the background color of the shell. For this participants had to write a function to convert the color components of a color value specified in the RGB color space, which are represented in hexadecimal notation, to decimal notation and vice versa. An example of conversion could be:
#FF32AC $< - >$ *[255,50,172].*
We did not expect this programming task to have an effect on participants' ability to fuzz a parsing function, but it was necessary to keep the effort and thus fatigue similar in both groups.

We conducted several pilot studies to judge the time needed for the TT and TP programming tasks and picked the four-hour time frame based on these.

7.2.2 Choice of Fuzzer

We chose libFuzzer since it was used by Plöger et al. This allowed us to draw from their experience and compare our results to theirs. libFuzzer also better fits our treatment task since we wanted to avoid participants trying to fuzz the target program's executable instead of only the parsing function.

7.2.3 Fuzzing Task

In the fuzzing task, the participants of the own-group were asked to fuzz the parsing function they had just programmed. The participants in the other-group were randomly assigned the code from one of the own-group. Due to having programmed the shell, they were familiar with the basic structure of the project, but they were unfamiliar with the parsing code - a similar situation as would be the case if a software engineer tests the code of a colleague in the same project. Participants of the Other-Group also received a short summary of the parser programming task. The summary included the actual task and all descriptions of edge cases and annotations.

As in Plöger et al., we gave our participants a time limit of ten hours in a time frame of ten days. Moreover, we anticipated a comparably high drop-out rate as in Plöger et al.'s first study (69%), and thus opted for the same support options that they did in their second study. Therefore, we structured the tasks.

We did this by adding six subtasks. The first two subtasks slightly differ between the two groups due to the fact the Other-Group had to work with the code of another person. The two subtasks that differ between the groups are marked with an * for the Own-Group and a + for the Other-Group.

1* Briefly familiarize yourself with libFuzzer.

1+ Briefly familiarize yourself with the new shell extension and with libFuzzer.

2* Select a suitable function to fuzz from your extension.

2+ Select a suitable function to fuzz from the new extension.

3 Write the fuzz target.

4 Compile and instrument all necessary code.

5 Start the fuzzer and interpret the outputs.

6.1 If you find a bug, fix it and continue fuzzing.

6.2 If you don't find a bug, try to improve your fuzzing.

We encouraged the participants to follow the subtasks. However, it was not mandatory to do so.

A detailed discussion on how to solve the fuzzing task can be found in Appendix D.1.

7.2.4 Support System

We also adopted the second measure, the support system, to counter the high dropout rate as well as to gain more insights into specific parts of the fuzzing process with libFuzzer. The support system is similar to the one used in Tiefenau et al. [181].

In the task description, participants were informed that they could contact the study assistant via e-mail for support if they got stuck - but after having made reasonable efforts to solve the problem. We responded to every support request within 24 hours but within the constraints of the typical working behavior of the study assistant. We followed the following scheme when responding to participants' questions.

If the question was a general technical question or a question regarding the understanding of the task, it was answered immediately by providing a solution for the problem.

If the question was a question regarding the fuzzing process and the fulfillment of the task, we delayed our answer by six hours. This was done to prevent the support system from becoming an easy way out of any problem while still being responsive enough not to lose participants or delay them too much. Follow-up questions were again answered immediately. We distinguished between two levels of support.

- Hint: The first type of support a participant received was a hint. A hint was a peace of information where the participant was told which specific thing in the fuzzing process could be helpful to further investigate in order to be able to solve the problem. An example of a hint could be: "It can be helpful for you to take a closer look at the instrumentation in subtask 4."

- Help: If the participant was not able to solve the issue after receiving a hint, help was given. This consisted of the solution to the problem with some information about the context. An example of help could be: "You need to instrument the target program. You can do this by adding the following parameter "-fsanitize=fuzzer-no-link" to the make command."

We also used the support channel to request participants work on subtasks if they handed in their solutions without having completed them. For subtask 6.2, "If you don't find a bug, try to improve your fuzzing." we regarded the usage of at least one sanitizer, such as ASAN, a corpus, and the checking of coverage or coverage visualization as having completed the sub-task. We have chosen these improvements because the usage of sanitizers and corpora are part of the fundamentals in the libFuzzer documentation and the question "How good is my fuzzer?" is answered in the documentation by checking the code coverage.

When evaluating the fuzzing process, we scored each subtask separately and we counted any intervention in the form of hint, help, or asking the participant to further work on a specific part of the task as a fail. We rated the overall fuzzing process only as a success if participants did not require any hint or help.

The two main benefits of this support system are

1) We gain more direct insights into problems since they are communicated to us in-situ and

2) we can gather more insights into all fuzzing subtasks as we hope that we don't lose as many participants as Plöger et al. did in their first study.

7.2.5 Scoring

As Plöger et al. in their second study, we implemented a fuzzing score to allow a more precise evaluation of our participants' issues working with libFuzzer, which enabled us to do a quantitative evaluation of the steps themselves. A point could be awarded for each of the following actions:

- Selecting a suitable function to fuzz.
- Write a buildable, meaningful, and triggerable fuzz target.
- Build all necessary code.
- Instrument all necessary code.
- Ran the fuzzer.
- Used a meaningful corpus.
- Used the address sanitizer on all code of interest.

The first five actions are a direct consequence of the subtasks and are the basics of implementing a reasonable form of fuzzing. We define building all necessary code as compiling and linking it. We also added the usage of a meaningful corpus and of a sanitizer since these were also fundamentals based on the libFuzzer documentation. Consequently, participants were able to get a fuzzing score between zero and seven points. However, participants only received a point if they did not get any hints or help for that action or any action leading up to it. A detailed discussion of the dependencies for the fuzzing process can be found in Appendix D.2. Therefore, the participant score reflects the score the participant would have received without support.

Successful Task Completion

Moreover, we adopted the binary success value used by Plöger et al. Participants successfully finished the fuzzing task if they found a bug using the fuzzer and also recognized it as a bug. Unlike Plöger's studies, our study could contain the situation that the parsing code being fuzzed contains no bug. We thoroughly discussed this situation but were unfortunately not able to find a fitting solution for that problem. Thus, we did not include all participants that could not find a bug in the evaluation of the success.

7.2.6 Recruitment & Participants

We recruited our participants from a course about system-level programming. The course is part of our universities bachelor's degree program in computer science. The course is scheduled for the third semester but can also be taken in the first semester or later on. Moreover, it is not mandatory to take the lecture, so only students who want to learn C and assembly take it. This is slightly in contrast to the recruitment of Plöger et al.'s first and second studies, where they both times recruited from a usable security master's course in computer science.

The study was announced at the end of the course as a code analysis study. The announcement was made in the lecture, on the mailing list, on the lecture website and in the exercises courses. At the time the study was conducted in the semester, 97 students were still taking part. Forty-nine students expressed interest in participating in the study, of which 45 answered the pre-questionnaire and were admitted to the study, which is 46% of the course participants at that time.

We compensated our participants with 210€ for participating in the study, which is an hourly rate of at least 14€. This was 4€ per hour more than the hourly rate of a student job at the university and 4.5€ more than the country's minimum wage at that time.

The demographics of our group can be seen in Table 7.2.

Gender	Male: 37	Female: 8	Other: 0	No Answer: 0
Age	min: 19, max: 30	mean: 21.84, median: 21	sd= 2.49, NA= 1	

TABLE 7.2: Participants' Demographics

Proficiency and Fuzzing Experience

The median of the self-reported proficiency in C and Linux was 4 on a scale from 1, very bad, to 7, very good.

Only 4 of the 45 participants had prior experience in the usage of a fuzzer. Three participants stated that they had used the fuzzer AFL or AFL++ before, while one participant used libFuzzer before. One participant did so at work, two at university, and one in private. To avoid an imbalance, the four participants with prior experience were split equally between the own and other group and the own code was assigned to the other pair.

Work Environment

Since Plöger et al. reported that some of their participants struggled to work on a fuzzing server without a GUI. Thus, We provided a pre-configured virtual machine where libFuzzer and the shell were already installed on the system and we could log their steps. Our participants had root access to the virtual machine and were free to use any software they liked. We used the latest LTS version of Ubuntu and libFuzzer (11.1.0) at the time the study was conducted.

7.3 Ethics

Our study was reviewed and approved by our institutional review board. The study complied with the EU General Data Protection Regulation. We have paid great attention to making sure that the code which other participants analyzed had no information to identify the author of it.

7.4 Limitations

Our study has the following limitations.

Participants As Plöger et al., we recruited our participants from a university course in computer science. While this is a legitimate user group, they do not generalize to all developers. However, Naiakshina et al. [135] found that students can be a helpful substitute for professional developers. Considering that it would have been next to impossible to run a 14-hour controlled experiment of this size with professional developers, we think this is an acceptable limitation. Nevertheless, of course, no generalized statements can be made.

Sample Size Despite this being the largest study of its kind, the sample size is still fairly small. Therefore, the statistical tests are subject to uncertainty.

Fuzzing Experience The majority of our participants had no previous fuzzing experience and must be considered fuzzing novices and not representative of specialized experts. However, this is a useful group to study since many organizations cannot afford specialized security testers or fuzzing experts, and thus, we think understanding the issues novice developers face and improving usability for them is a worthwhile goal.

Fuzzer Selection Our participants only worked with the fuzzer libFuzzer. For other fuzzers, the situation may be different.

Guiding the Participants with Subtasks We decided to support our participants by breaking down the overall task into structured subtasks and thus providing guidance within the process of fuzzing. This was done with the intention of being able to gather more information, especially in the later steps of the fuzzing process, which the study of Plöger et al. lacked. However, this type of guidance does not fully reflect the situation a user is in outside of this study.

Support System We implemented a support system to overcome the problem of participants getting stuck in the early stages of the fuzzing process and not delivering many insights into problems in later stages. We adjusted the scores, so the support process did not affect the quantitative scores, while at the same time offering more insights.

7.5 Results

Of the 45 participants who started the study, 38 participants finished their programming task (either treatment or control) successfully. Two did not manage to create functional code, and 5 dropped out. 40 participants started the fuzzing task and 34 participants finished it. Participant P35 worked on the fuzzing task, submitted their work, and was asked to further work on the task as they did not work on all subtasks and had time left, although they received multiple hints and help in that process. However, they did not feel comfortable further working on the task and decided to drop out. We offered full compensation when participating in the final interview, which they agreed upon. We also report the results of P35 for the scores and in the qualitative analysis because, at the time of their dropout, they could not have received any more points for the score. Of the 5 participants who did not have a partner at the end, 2 were fuzzing their own code and 3 others' code. Consequently, we quantitatively evaluated the data of 14 pairs. This results in a completion rate of 75.6%. This is more than double that of Plöger et al.'s 31%. This leads us to believe the modifications we made to the study protocol are beneficial.

We asked all of the 11 participants who dropped out what their reason was. Three did not answer, five stated lack of time, and two reported illness. Only one participant dropped out, stating that they did not feel able to continue despite having received multiple hints and help.

We analyzed our data both quantitatively as well as qualitatively. For the quantitative analysis, our main hypothesis was that the independent variable code authorship - own vs. other - has an effect on the fuzzing outcome. For all our statistical tests, we use two-tailed non-parametric tests, and we report effect sizes, p-values, and confidence intervals with an α of 0.05.

7.5.1 Programming Task Results

We manually checked all 40 submissions of the programming tasks for functionality. Nineteen participants started in the control task, the color converter, and all handed in a functional solution. This code was then discarded as it was not needed any further.

Twenty-one participants started the treatment task, the parser. Two did not manage to create a functional solution in the four-hour time period. Their code was discarded and they were removed from the own-group since the lack of functional code meant that they would not be able to fuzz their own code. Nineteen handed in a functional solution. The 19 participants from the other-group were randomly assigned codes from these 19 solutions.

Before we started the fuzzing task, we analyzed the 19 parser codes in-depth to make sure the code was suitable for the study. While we expected most parsers to contain bugs[1], we needed to check that we had a mix of code with and without bugs for our study and that the bugs were easy enough to find in a ten-hour time frame.

To do this, we fuzzed the code ourselves. We also used a small custom corpus. We found bugs in 15 of the 19 parsers within seconds. The four solutions that did not crash quickly were then examined more thoroughly. First of all, we checked whether all code was covered by our fuzz target, which was the case. Then we created a fuzzing campaign including fuzzing with no sanitizers and with ASAN, MSAN, and UBSAN and value profiles in parallel, which also included our corpus. We ran the campaign for 24 hours but were not able to find any bugs during that time in those four parsers. This gives us a nice mix of parsers with and without bugs.

The two participants, P1 and P22, who did not successfully complete the programming task and were removed from the main experiment were assigned to fuzz a parser from the own-group to let them finish the study and gain qualitative insights. They were not used for any of the statistical tests.

So at the end of the programming task, we had 19 participants starting the fuzzing task on their own code and 19 participants starting the fuzzing task randomly assigned code from the own-group. We also had P1 and P22 fuzzing outside of the main experiment.

7.5.2 Support System

In the fuzzing task, we gave 45 hints in total, spread over 24 different participants (53%). Only three participants who received a hint dropped out of the study.

We also gave Help to participants 18 times, spread over 7 participants in total. Of the participants that received Help, only one participant dropped out of the study.

Overall, only 3 participants who benefited from the support system dropped out of the study.

The distribution of Hints and Help given for the participants who fuzzed their own code and the participants who fuzzed code from a different person and for the specific steps are shown in Figure 7.1. The high number of hints and help in the working fuzz target, building, and instrumentation steps are particularly striking. Moreover, the hints for interpreting crashes are fairly high.

[1]Writing parsers is hard

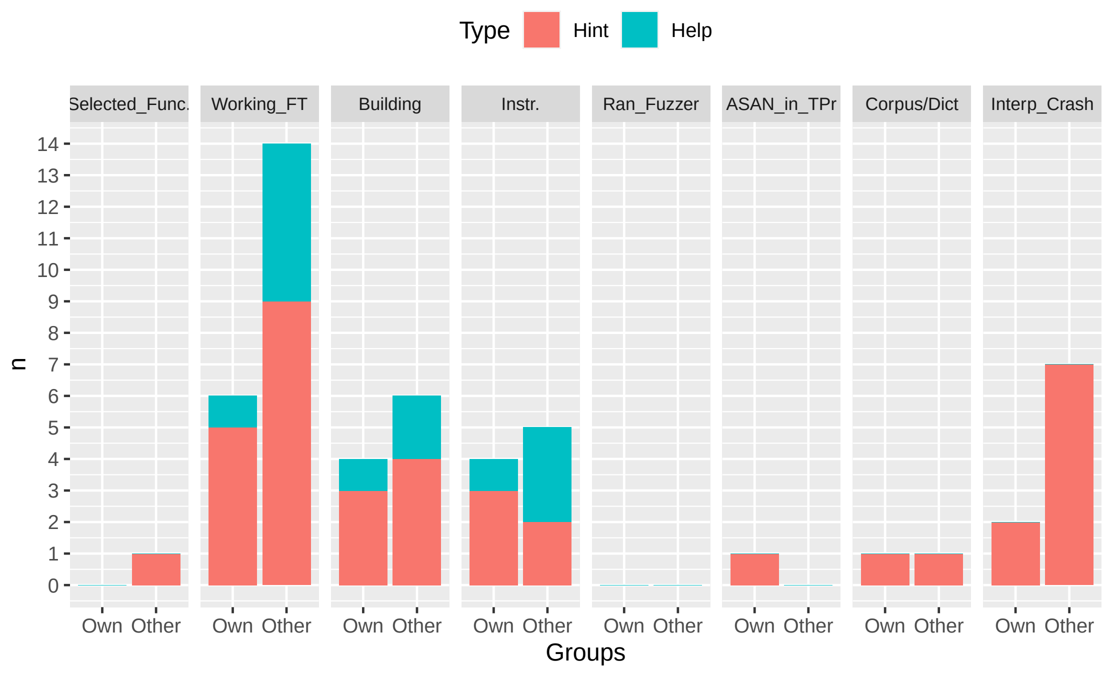


FIGURE 7.1: Hints and Help given per steps for the fuzzing task

We will get into more detail about that as well as the differences between the two groups in the later stages of our analysis.

7.5.3 Success Rate

The corresponding table for the success in the fuzzing task for our 11 pairs that could find a bug can be seen in Table 7.3. We had no pair where both members were successful. We had five pairs where both members failed. While in four pairs, only the participant fuzzing their own code was successful, and only in two pairs only the participant fuzzing the other's code was successful. This leads to an OR of 2. However, this result is not statistically significant, and the CI is very wide [0.33, 7].

		Own	
		success	no success
Other	success	0	2
	no success	4	5

TABLE 7.3: Comparison of pairs for success

Participant	Sel. Func.	Working FT	Building	Instr.	Ran Fuzzer	Sanitizer used	Corpus/Dict	Score	Trig. UEIC	Fixed UEIC	Trig. bug	Success
Own												
4	1 / ✓	✓/ ✓/ ✓+	✓+	✓+	✓	✓	✓	3	✓	-	✓	✓
6	1 / ✓	✓/ ✓/ ✗	✓	✓	✓	✓	✗	5	✗	✗	✗	-
7	1 / ✓	✓/ ✓/ ✓*	✓	✓	✓	✓	✓	6	✗	✓*	✓	✓
15	1 / ✓	✓/ ✓/ ✓	✓	✓	✓	✓	✓	7	✓	-	✓	✓
23	1 / ✓	✓/ ✓* / ✗*	✗	✗	✗	✗	✗	1	-	-	-	-
28	1 / ✓	✓/ ✓/ ✗*	✓*	✗*	✓	✓	✓	3	✗	✗	✗	-
30	5 / ✓	✓/ ✓/ ✓	✓	✓	✓	✓	✓	7	✗	-	✓	✓
32	5 / ✓	✓/ ✓/ ✓	✓	✓	✓	✓	✓	7	✗	✓	✓	✓
33	5 / ✓	5 / ✓/ ✓	✓	✓	✓	✓	✓*	6	✗	✓	✓	✓
34	1 / ✓	1 / ✓/ ✓	✓	✓	✓	✓	✓	7	✓	-	✓	✓
37	1 / ✓	✓/ ✓/ ✗	✓	✓	✓	✓	✓	6	✗	✗	✗	-
43	2 / ✓	✓/ ✓/ ✗	✓	✗	✓	✗	✓	4	✗	✗	✗	-
Own - with no bug												
16	1 / ✓	✓/ ✓/ ✓	✓	✓	✓	✓	✓	7	✗	✓	-	-
17	1 / ✓	1 / ✓/ ✓	✓	✓	✓	✓	✓	7	✗	✓	-	-
39	1 / ✓	✓/ ✓/ ✓*	✓*	✓	✓	✓*	✓	3	✗	✓*	-	-
40	1 / ✓	✓/ ✓/ ✓	✓	✓	✓	✓	✓	7	✗	✓	-	-
Other												
3	1 / ✓*	✓/ ✓/ ✓*	✓	✓	✓	✓	✓	4	✗	✓*	✓	✓
5	1 / ✓	✓/ ✓/ ✓+	✓+	✓+	✓	✓	✓	3	✓	-	✓	✓
8	1 / ✓	✓/ ✓/ ✓	✓	✗	✓	✓	✓	7	✓	-	✓	✓
9	1 / ✓	✓/ ✓/ ✓	✓	✗	✓	✓	✓	7	✗	✓	✓	✓
11	1 / ✓	✓/ ✓/ ✓	✓	✓	✓	✓	✓	7	✗	✓	✓	✓
12	1 / ✓	✓/ ✓/ ✗	✓	✓	✓	✓	✗	5	✗	✓	✗	-
13	1 / ✓	✓/ ✓/ ✓*	✓	✗	✓	✗	✓	4	✗	✓*	✓	✓
19	1 / ✓	✓/ ✓/ ✗	✓*	✓	✓	✓	✗	3	✓	✗	✗	-
20	1 / ✗	✓/ ✓/ ✓+	✓	✗	✓	✗	✗	3	✗	✓*	✓	✗
21	1 / ✓	✓/ ✓/ ✗	✓	✓	✓	✓	✓	6	✗	✗	✗	-
29	1 / ✓	✓/ ✓/ ✓	✓*	✓	✓	✓	✓	5	✗	✓	✓	✓
31	1 / ✗	✓/ ✗/ ✗	✗	✗	✗	✗	✗	0	-	-	-	-
44	1 / ✓	✓/ ✓/ ✓*	✓	✓	✓	✓	✓*	5	✗	✓*	✓	✓
Other with no bug												
36	1 / ✓	✓/ ✓/ ✓+	✓	✓	✓	✓	✓	6	✗	✓	-	-
41	1 / ✓	✓/ ✓/ ✗	✓	✗	✓	✗	✓	4	✗	✗	-	-
10	1 / ✓	✓/ ✓/ ✓*	✓	✓	✓	✓	✓	6	✗	✓*	-	-
RC Task - Unusable programming solution												
22	1 / ✓	✓/ ✓+ / ✓+	✓+	✓+	✓	✓+	✗	1	✓	-	✓	✗
1	1 / ✓	✓* / ✓/ ✗+	✓*	✓*	✓	✗	✓	2	✗	✗	✗	-

TABLE 7.4: Results of All Participants for the Fuzzing Task: ✓: success - ✗: failure - green: success without support - yellow: success with support - blue: success after support in previous step - red: no success - orange: prior knowledge of fuzzing

7.5.4 Fuzzing Steps

As described in section 7.2.5 we also computed a more detailed stepwise score of the fuzzing task. The results can be found in Table 7.4.

In the table, success is denoted with ✓, while failure is denoted with ✗. We have divided success into three levels to specify further how our participants succeeded in the different fuzzing steps. The resulting five outcomes are:

green: The color green is used when a participant was successful in this and all dependent steps without getting any hint or help. This is equivalent to receiving a point.

blue: The color blue is used if a participant succeeded in a step without any hint or help in that step but received a hint or help in a dependent step before. The participant did not receive a point in this step since they would not have reached this step without our support. However, the blue highlights that the step itself did not cause the participant a problem, which is relevant to the usability evaluation of the fuzzer.

yellow: If a participant succeeded in a step but received a hint or help in it, it is colored in yellow.

red: A step is marked in red if the participant did not succeed.
gray: The color gray is used if a step could not be positively or negatively been rated.

When we gave a hint, it is shown with an asterisk, * . For help, we used the plus sign, +.

The Table 7.4 is structured as followed:

Selected function: The column *Sel. Func.* is twofold and depicts on the left side the number of functions a participant tried to fuzz. On the right side, it is shown whether the participant chose a meaningful function to fuzz in the context of the task.

Working fuzz target: *Working FT* is tripartite. at first, it is shown whether the fuzz target was *buildable*, followed by being *meaningful* and *triggerable*. A fuzz target is *buildable* if commands exist so it can be built, or, in other words, if it is syntactically correct. A *meaningful* fuzz target is a fuzz target where it is tried to fuzz a function that is reasonable with respect to the task. It has to be a function of the target program. A *triggerable* fuzz target is able to repeatedly trigger a bug in the target program.

Building: The column *Building* shows whether the participant was able to build all necessary code,

Instrumentation: while in *Instr.* it is shown whether the participant was able to instrument all necessary code.

Ran fuzzer: Whether a participant *Ran Fuzzer* is displayed in the corresponding column.

Sanitizer used: The usage of sanitizers, most likely the address sanitizer, is presented in the column *Sanitizer used*.

Corpus or dictionary: Analogously, this applies to the column *Corpus/Dict*.

Score: The score of each participant, as specified in Section 7.2.5 can be seen in the last column before the section.

Triggered UEIC: The first column *Trig. UEIC* (User Error Induced Crash) after the section shows whether the participant triggered a user error induced crash (UEIC). These are not true program crashes but are crashes that are caused by the user, for example, due to programming mistakes in the fuzz target.

Fixed UEIC: In *Fixed UEIC* it is shown whether the participant was able to overcome the UEIC.

Triggered bug: Whether a participant triggered a bug in the target program is depicted in the column *Trig. bug*

Correctly interpreting bug: The last column shows whether the participant found a bug in the target program and also recognized it as a bug, which is the equivalent to "*success*" in the studies of Plöger et al.

7.5.5 Fuzzing Scores

Scoring Procedure When we aggregated the data, we realized that due to the large amount of information per participant and the complexity of the task, scoring the participants could be error-prone. Therefore, we decided to score each participant by 3 researchers independently and in parallel. We started with 10 % (4) of our participants and compared all 3 scorings afterwards. Possible differences were resolved through discussion. This was done to ensure that all researchers had the same concept for the scoring. We repeated the process again with 4 participants. After that, it became clear that all 3 researchers had the same concept so we scored the remaining participants in one go. In summary, every participant was scored by 3 researchers independently and in parallel, and possible differences were resolved through discussion.

The fuzzing score of a participant could lie between 0 and 7. For the 14 pairs, the mean of the fuzzing score for the own-group is 5.14, and for the other-group, it is 4.71. This results in a difference in means of 0.43, CI [-0.91, 1.76], p = .48 (Wilcoxon signed rank). Similar to the above, we can detect no statistically significant difference. In addition, the mean difference of 0.43 is fairly small, which means there would also be little practical difference, even if it was statistically significant.

Participants with prior experience Participants P4 and P44 as well as P30 and P19 reported prior knowledge in fuzzing C/C++ code. They are depicted in orange in Table 7.4. Despite this, their results were not better than the results of the other participants. On the contrary, if anything they performed worse.

7.5.6 Crash Handling

The results of how our participants dealt with crashes produced by the fuzzer can be seen in the second part of Table 7.4. The color scheme and character selection follow the same principle as in the first part of the table.

Overall, we did not find any particular differences between the two groups when it came to handling crashes, which we found surprising since it is plausible that knowing the code in which the bug is contained might give an edge.

Crash Output: Four things in the output of the fuzzer have left a lasting impression on our participants.

Type of crash: The type of crash was mentioned by nearly all of our participants. They felt it helped to understand the situation and they were thus pleased with the information.

Crash file: The crash file or the crashing input respectively was also mentioned by many participants as one of the information they received from the fuzzer. The crashing input was actively used to better localize the problem's cause. However, there was also some confusion regarding crashes that were caused by the empty input.

> I got the crash file, but it was empty. I relied on that. I have no idea what the reason was. That also bothered me a bit. (P23 of own group)

Stack trace: If the participants used the address sanitizer, the stack trace was heavily used to localize the crash and it was rated very positively by our participants. In this, the crashing function, the line of code and the sequence of called functions were highlighted as very useful.

Heap info: The heap information provided by the address sanitizer was the last of the four outstanding provided information. Although this was more of a negative example, as nearly nobody was able to use it effectively, but at least it looked pretty:

> There was a nice colored diagram of the memory with the bytes and how far they're used and stuff. And which ones are partially accessible and so on. Looks nice, but didn't help me much. (P43 of own-group)

Problems of Interpretation

Nevertheless, several participants had severe problems effectively using the information provided to determine the cause of the crash and how to fix it. Participant P43 of the own-group answered the question of what the fuzzer provided to fix crashes:

> For the most part, lots of crap. That was my big problem, I must say. The fuzzer detected bugs wonderfully. How to solve these bugs was very unclear to me. And that was my biggest problem with the fuzzer. (P43 of own-group)

Fixing Crashes

The user error-induced crashes were a very noticeable problem for our participants. Of the 34 participants who ran the fuzzer, 26 triggered a UEIC and only ten of those 26 were able to find and fix it without any support. Accordingly, this was a part where support was regularly sought. We were not able to find meaningful differences in either the frequency of UEIC or the success of fixing them between groups.

Moreover, noticeable differences between trying to fix UEICs and real bugs in the program under test could not be found, which is not very surprising since the fuzzer can not distinguish between them and thus presents the same type of information for both.

Fifteen of our participants reached the stage of trying to fix the bugs found by the fuzzer. Nine of the 15 participants were able to remove all bugs the fuzzer had found. However, one of the participants actually cut the functionality of the program in the process. It is also noteworthy that three participants, P8 and

P9 of the own-group, and P30 of the other-group, altered the code of the target function to overcome problems caused by a bad fuzz target. This resulted in crashes or leaks when running the shell or the inability to compile it alone. But the code was functional with the respective fuzz target.

Overall, the remaining participants were fairly successful in removing the bugs reported by the fuzzer. Moreover, no difference can be seen between the own-group participants and the other-group participants.

7.5.7 System Usability Scale

In the post-questionnaire, we asked our participants to fill out the System Usability Scale (SUS) to determine the usability of libFuzzer. The own-group had a median of 61.25 and mean of 59.69, while the other-group had a median of 45 and mean of 47.94. A Wilcoxon ranked sum test just misses statistical significance, p=0.051. However, a more than 15-point difference in the median and more than a 10-point difference in means of a SUS score is practically relevant. For more information on interpreting p values and effect sizes, please see Cumming and Finch [46]. To determine the effect size, we again used the difference in means. The difference in means showed an effect size of MD = 11.75, CI [0.26, 23.23]. It is noteworthy that this CI is statistically significant. However, this is down to the bootstrapping method used to compute the CI. This tends to produce smaller confidence intervals [55], which causes the confidence interval not to include 0 even though the Wilcoxon ranked sum test does not show statistical significance. We interpret these results to mean that the population average can range from no impact to a positive impact for those who fuzz their own code.

However, it is even more noteworthy that even the upper score of 61.25 is a pretty bad SUS score, as described by Jeff Sauro [165]. This would be the equivalent of an F in a school grading system. Considering the context that our participants had just completed a semester voluntarily learning C and assembly, these scores are particularly damming and highlight that fuzzers have much potential for improvement for this user group. To add some more insights into this, we conducted a qualitative analysis of the diaries and interviews.

7.6 Qualitative Analysis

We used inductive coding with two researchers to qualitatively analyze the participants' interviews and their diaries. We adopted the coding process of Plöger et al. We randomly divided the 35 interviews into twelve sets of three interviews each. Starting with the first set, the two researchers coded the three interviews of the set independently and in parallel. The resulting codes and codebooks were discussed, changes were incorporated and they agreed on a common codebook. The changes were applied to all interviews coded so far. The resulting codebook was used to code the interviews of the next set. This process was repeated until all interviews were coded.

7.6.1 Familiarization with New Code

The participants of the other-group described the familiarization with the new code at first as not very difficult. To get to know the code, they read the code but also played with the program to see how it behaved with different inputs. Although, some of them specifically stated that they did not take a closer look at the code at this point, as they anticipated that they had to dive in deeper in the later stages of the task.

7.6.2 Selecting the Function to Fuzz

The selection of the target function did not pose a problem for almost all participants. For our participants of the own-group, the by far most common reason for choosing a function was that the function was the only self-written function or that it was the highest self-written function in terms of code depths.

The reason for choosing the only self-written function was also sometimes the reason to choose a function for the participants of the other-group. Even though they did not write the chosen function themselves, they recognized the function as the function of the programming task of the own-group and thus chose it. However, their most common reasons for choosing a function to fuzz were working directly with the user input.

7.6.3 Writing the Fuzz Target

Principally, participants in both groups showed a sufficient understanding of what a fuzz target is. Only the two participants, P1 and P22, who had also solved the RC programming task in such a way that it could not be used for fuzzing, and the participants P23 and P31, did not have a sufficient understanding.

Looking at the fuzzing process chronologically, only very few of our participants reported problems in this stage of the fuzzing process. This was mostly the case because participants of both groups were able to write fuzz targets that were *buildable* and *meaningful*. The few problems actually referred exclusively to using the parameters of the LLVMFuzzerTestOneInput function and then adapting them to the target function.

7.6.4 *Triggerable* Fuzz Target

Creating a *triggerable* fuzz target is the step with the most pronounced difference between the groups and also the first step many participants struggled with. This was a severe problem for many of our participants resulting in only 12 of our 34 finished participants to fulfill that task on their own. There were noticeable differences between the two groups in this regard. More than twice as many participants from the own-group were able to solve this step without support than from the other-group.

Many of our participants made mistakes when passing strings between the fuzzer and the target program resulting in UEICs. P39 of the own-group stated on the matter:

> ... that scanRC sets initial conditions... that the output did not end on the terminal symbol. This also led to errors for me, because I assumed this in the scanRC function.

7.6.5 Building & Instrumentation

The building and instrumentation were the second major problematic step our participants encountered. In contrast to the step of writing the fuzz target, we were not able to find striking differences between both groups.

It should be said at the outset that some of the participants left the instrumentation almost completely out of the equation. Only two participants had a sufficient understanding of what instrumentation is. The other participants either tried to gather information about it but failed or admitted that they read the word but ignored it completely. Participant P21 of other-group said:

> I honestly haven't really thought about it that much. I thought that you just somehow prepare it in such a way that it all works together.

The participants clearly focused on putting the right flags at the right position in the Makefile or in the command executed on the command line. A result of that was that some participants only instrumented the fuzz target and not all other necessary code since this seemed to be the easiest way of achieving the compilation. Frequently, participants could not compile the code by guessing and went on to adjust the code to solve the problem. The most popular way was to remove the main function of the program under test.

> Because I have found no code example for the application of this fuzzer-no-link. To make it simple, I commented out the main function and then simply compiled everything with libFuzzer. That worked, but then the code is no longer executable. And you have to comment out the main function again and again when you are done with fuzzing and then compile again. (P39 of own-group)

The struggle of our participants also resulted in the need for hints and help.

7.6.6 Running the Fuzzer

The groups did not differ in the perceived information that libFuzzer displayed to them. The most noticed information was the number of all runs, the event codes, and the coverage. Most of the time, however, the coverage was used to determine if the fuzzer was running properly. The other markers for a properly running fuzzer were that the fuzzer runs at all and the occurrence of crashes. Overall, there was a slight uncertainty as to whether the fuzzer was actually doing useful things. Participant 36 of the other-group stated:

> As far as that is concerned, I wasn't sure in the end. Even now, I'm still not sure if the fuzzer really ran correctly.

7.6.7 Crash Handling

The results of how our participants dealt with crashes produced by the fuzzer can be seen in the second part of Table 7.4. The color scheme and character selection follow the same principle as in the first part of the table.

Overall, we did not find any particular differences between the two groups when it came to handling crashes. This was slightly surprising since it had seemed plausible, that knowing the code where the crash happens would be beneficial.

7.6.8 Problems of Interpretation

Nevertheless, several participants had severe problems effectively using the information provided to determine the cause of the crash and how to fix it. Participant P43 of the own-group answered the question of what the fuzzer provided to fix crashes:

> For the most part, lots of crap. That was my big problem, I must say. The fuzzer detected bugs wonderfully. How to fix these bugs was very unclear to me. And that was my biggest problem with the fuzzer. (P43)

7.6.9 Fixing Crashes

The user error-induced crashes were a very noticeable problem for our participants. Of the 34 participants who ran the fuzzer, 27 triggered a UEIC and only ten of those 27 were able to find and fix it without any support. Accordingly, this was a part where support was regularly sought. We were not able to find meaningful differences in either the frequency of UEIC or the success of fixing them between the own- and other-group.

Moreover, noticeable differences between trying to fix UEIC and real bugs in the program under test could not be found, which is not very surprising since the fuzzer can not distinguish between them and thus presents the same type of information for both.

Fifteen of our participants reached the stage of trying to fix the bugs found by the fuzzer. Nine of the 15 participants were able to remove all bugs the fuzzer had found. However, one of the participants actually cut the functionality of the program in the process. Surprisingly, the code coverage information revealed indications for dead code for the code of four participants, which proved to be true. The participants did not detect this. It is also noteworthy that three participants, P8 and P9 of the own-group, and P30 of the other-group, altered the code of the target function to overcome problems caused by an insufficient fuzz target. This resulted in crashes or leaks when normally running the shell or the inability to compile it, but the code was functional with the respective fuzz target.

Overall, the remaining participants were fairly successful in removing the bugs reported by the fuzzer. Moreover, no difference can be seen between the Own-Group participants and the Other-Group participants.

7.6.10 Perceived Difficulty of Fuzzing Other Peoples' Code

In the interviews, it was noticeable that none of the participants in the other-group associated the difficulties they encountered with fuzzing with the fact that they were fuzzing other people's code.

Also, when asked if it would have been easier to fuzz their own code, the other-group showed a spread of opinions. On the one hand, some of the participants, for example, P44, stated that fuzzing their own code would have been easier.

> Yes, so for sure it's easier. You know your own code, so you don't have to deal with how it works.

On the other hand, There was a broader movement within the other-group that would not have thought this would be easier. One of them was participant P21:

> Probably not. The understanding of the code itself was not the problem for me personally.

In contrast to that, the own-group was uniformly sure that it would be much harder to fuzz code from someone else. Participant P8 said:

> I think so. Because I had such a good understanding of my code. [...] But I think it would have been more difficult [to fuzz some elses code].

7.7 Discussion

7.7.1 Own Code vs. Others' Code

When we take a look at the situation from a practical point of view, we did not find any statistically significant differences between the group that fuzzed their own code and those who fuzzed the other participants' code. However, it must be noted that while we recruited as many participants from the systems programming course as possible, our sample is still fairly small. Thus only large effects or even very large effects are likely to be statistically significant. The mean difference in the fuzzing score, however, was fairly small, only 0.43 on a 7-point scale. So even if this were statistically significant, it would have little practical implication. The odds ratio for success of 2 in favor of fuzzing one's own code was also not statistically significant but would be a more meaningful effect.

This might be attributed to the fact that the creation of a working fuzz target seemed to be easier for the participants that fuzzed their own code. Moreover, small advantages could be seen for the recognition of bugs, also for participants of the own-group.

Interestingly, the biggest difference could be seen in the participants' usability assessment of libFuzzer. Here the own group gave libFuzzer an 11-point better SUS score than the other group. This is interesting because the fuzzer was the same for both groups, but obviously, the experience was a different one.

To summarize, we do not have robust evidence that either group has a statistical or practical advantage over the other. But it does look like that the own group might have a slight edge. This, however, needs to be tested again in future work.

So in the debate about which group (own vs other) is better suited to fuzz code, we would currently recommend adopting the position that for our CS students it doesn't matter, but it does matter that someone does.

However, it is also clear that the usability of libFuzzer needs to be improved, as can be seen in the low SUS scores and the many problems both groups had.

7.8 Summary

In this paper, we present the first controlled experiment examining the effect of fuzzing code written by oneself or by others. We conducted our study with 45 CS students, of whom 34 made it all the way through. Our results show little difference between the two groups. However, there are some indications that the own group might have a slight advantage. We plan future studies to examine whether these differences can be substantiated.

But we also highlight that libFuzzer - the fuzzer we used in our study - poses significant challenges for fuzzing novices, and we call on the community to improve fuzzing usability.

Chapter 8

Conclusion

Although fuzzing has good success in large companies and has been a hot topic in academia for several years, there had not been any studies on its usability. Therefore, I conducted four user studies to shed light on the usability perspective of fuzzing.

In the first study, I qualitatively evaluated the usability of the Clang Static Analyzer and libFuzzer with CS students and CTF players. In terms of fuzzing, I found several usability issues with libFuzzer. Furthermore, I found indications that CTF players, as a representative of hackers, had the same problems as the CS students. Although, they were better able to overcome those problems and dig deeper into the fuzzing process. Unfortunately, the study struggled with a comparably low number of participants. Moreover, the CS students failed fairly early in the fuzzing process, so I could only retrieve sparse insights into the later steps.

In the second study, I examined the usability of the two most popular fuzzers, AFL and libFuzzer, with CS students to get deeper insights into the later steps of the fuzzing process and to improve the insights of the previous study. The results showed that the CS students achieved better results with AFL and favored it from a usability perspective. Moreover, I was able to solidify the afore-found usability issues of libFuzzer and additionally found usability issues encountered in the later steps of the fuzzing process. Furthermore, I highlighted the positive aspects of both fuzzers.

To be able to make a more generalizable statement on the findings, I conducted a third study where I replicated the previously conducted study with freelance developers. The freelancers performed very similarly to the CS students and again achieved better results using AFL. Moreover, AFL was again more appreciated. The results showed that CS students are adequate substitutes for freelance developers and potentially professional developers.

Finally, I conducted a user study to provide first insights on whether it is more beneficial to fuzz someone's own code and code written by others. The results did not show any evidence that a practical difference exists between fuzzing someone's own code or the code of others.

Overall, I was able to derive the following insights from my studies:

Fuzzing Usability I propose the following recommendations extracted from the results of my studies to improve the usability of fuzzing so that the barrier to entry can be overcome more easily and that fuzzing becomes accessible to a broader audience.

- **Assist users in finding suitable functions to fuzz** While the CTF players could find meaningful functions to fuzz in a straightforward manner, the CS students and freelance developers struggled more in this step. Since it is essential for an efficient fuzzing run to choose the best possible functions to fuzz, it would be helpful to assist users in making a good decision. This can also lower the entry bar to start with fuzzing as it is one of the first steps to do and can therefore contribute to fuzzing being used by non-experts as well.

- **Fuzz target creation** In my study about the usability of a static code analysis tool and libFuzzer, I found that especially the non-expert participants struggled heavily with writing more than the most simple fuzz target. I concluded from this that it would be very helpful to support users in creating fuzz targets or improving the coverage-guided self-exploration. This recommendation was supported by my findings in comparing AFL and libFuzzer, where the participants were very vocal about the positive aspects of fuzz-target-less fuzzing. Therefore, I extend the recommendation to be able to fuzz arbitrary functions without much knowledge of fuzzing. However, this is, of course, an open research question.

- **Build automation** The automation of the building process would be highly desirable as the process itself can require a lot of manual work and, frequently, advanced knowledge about the interaction and connections of several components.

- **Sanitizer Usage** The way in which sanitizers should be applied was a point of discussion. At first, I recommended making it an opt-out option so that novice users would use it without thinking about it. However, in our later studies, I found that sanitizers can also be overwhelming, and an opt-in might be a more usable solution. Undoubtedly, sanitizers can be very beneficial and are all in all recommended for users to get more information on the fuzzer.

- **Error Messages** AFL provided meaningful error messages to the participants so that they could overcome problems and get deeper into the fuzzing process. Adopting those error messages is recommended.

- **UI Guidance** In the study about the comparison of AFL and libFuzzer, the replication and the comparison of fuzzing self-written code or code written by others, it has been shown that the guidance of the participants with subtasks has enabled them to reach deeper into the fuzzing process.

Therefore, it can be helpful if this type of guidance is integrated into the user interface of a fuzzer. Especially a graphical interface, which gives information about the fuzzing steps and, e.g., the use of sanitizers, can bring advantages. It can also be good if, for example, experts can opt out of the graphical interface.

- **Metrics of Fuzzing Runs** Even though some metrics by AFL and libFuzzer about the fuzzing run are provided, they did not help the participants assess whether the fuzzer was running properly. Developing more meaningful, usable, and easier-to-interpret metrics is an interesting open research question.
- **Crash Analysis** In our studies, user error-induced crashes happened very frequently. Participants had problems understanding the cause of those crashes and were often overwhelmed by getting to the core issue of a crash. It would be useful if fuzzers could recognize if a crash originated from a problem in the fuzz target and not from a bug in the target program.
- **Better documentation** This is not a very glamorous recommendation. Nevertheless, the documentation of AFL and libFuzzer was rated anything but good. Proper documentation can increase usability.

In conclusion, I hope that these initial findings on the usability of fuzzers will help non-experts use them more frequently and with fewer impediments and that others will take up my research and take it further.

Bibliography

[1] Y. Acar, S. Fahl, and M. L. Mazurek. "You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users". In: *2016 IEEE Cybersecurity Development (SecDev)*. 2016, pp. 3–8.

[2] Y. Acar et al. "You Get Where You're Looking for: The Impact of Information Sources on Code Security". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 289–305.

[3] Yasemin Acar et al. "Security Developer Studies with GitHub Users: Exploring a Convenience Sample". In: *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*. Santa Clara, CA: USENIX Association, July 2017, pp. 81–95. ISBN: 978-1-931971-39-3. URL: `https://www.usenix.org/conference/soups2017/technical-sessions/presentation/acar`.

[4] *AFL*. `https://github.com/google/AFL`. Accessed: November 2022.

[5] *AFL++*. `https://github.com/AFLplusplus/AFLplusplus`. Accessed: November 2022.

[6] Andrei Arusoaie et al. "A comparison of open-source static analysis tools for vulnerability detection in C/C++ Code". In: *Proceedings - 2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017* (2018), pp. 161–168. DOI: `10.1109/SYNASC.2017.00035`.

[7] Cornelius Aschermann et al. "Ijon: Exploring Deep State Spaces via Fuzzing". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1597–1612. DOI: `10.1109/SP40000.2020.00117`. URL: `https://doi.org/10.1109/SP40000.2020.00117`.

[8] Nathaniel Ayewah and William Pugh. "A Report on a Survey and Study of Static Analysis Users". In: *Proceedings of the 2008 Workshop on Defects in Large Software Systems*. DEFECTS '08. Seattle, Washington: Association for Computing Machinery, 2008, pp. 1–5. ISBN: 9781605580517. DOI: `10.1145/1390817.1390819`. URL: `https://doi.org/10.1145/1390817.1390819`.

[9] Domagoj Babić et al. “FUDGE: Fuzz Driver Generation at Scale”. In: ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 975–985. DOI: `10.1145/3338906.3340456`. URL: `https://doi.org/10.1145/3338906.3340456`.

[10] Titus Barik et al. “From Quick Fixes to Slow Fixes: Reimagining Static Analysis Resolutions to Enable Design Space Exploration”. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016, pp. 211–221. DOI: `10.1109/ICSME.2016.63`.

[11] Jason Bau et al. “Vulnerability factors in new web applications: Audit tools, developer selection & languages”. In: *Stanford, Tech. Rep* (2012).

[12] Al Bessey et al. “A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World”. In: *Commun. ACM* 53.2 (Feb. 2010), pp. 66–75. ISSN: 0001-0782. DOI: `10.1145/1646353.1646374`. URL: `https://doi.org/10.1145/1646353.1646374`.

[13] Paul E. Black, Vadim Okun, and Barbara Guttman. *Guidelines on Minimum Standards for Developer Verification of Software*. en. Oct. 2021. DOI: `https://doi.org/10.6028/NIST.IR.8397`. URL: `https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=933350`.

[14] William Blair et al. “HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing”. In: *Proceedings 2020 Network and Distributed System Security Symposium* (2020). DOI: `10.14722/ndss.2020.24415`. URL: `http://dx.doi.org/10.14722/ndss.2020.24415`.

[15] William Blair et al. “HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing”. In: *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. URL: `https://www.ndss-symposium.org/ndss-paper/hotfuzz-discovering-algorithmic-denial-of-service-vulnerabilities-through-guided-micro-fuzzing/`.

[16] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. “Fuzzing: Challenges and Reflections”. In: *IEEE Softw.* 38.3 (2021), pp. 79–86. DOI: `10.1109/MS.2020.3016773`. URL: `https://doi.org/10.1109/MS.2020.3016773`.

[17] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. “Boosting fuzzer efficiency: an information theoretic perspective”. In: *ESEC/FSE ’20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. Ed. by Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann. ACM, 2020, pp. 678–689. DOI: `10.1145/3368089.3409748`. URL: `https://doi.org/10.1145/3368089.3409748`.

[18] Marcel Böhme, Laszlo Szekeres, and Jonathan Metzman. “On the Reliability of Coverage-Based Fuzzer Benchmarking”. In: *Proceedings of the 44th International Conference on Software Engineering (ICSE ’22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[19] Guillaume Brat et al. “IKOS: A Framework for Static Analysis Based on Abstract Interpretation”. In: *Software Engineering and Formal Methods*. Ed. by Dimitra Giannakopoulou and Gwen Salaün. Cham: Springer International Publishing, 2014, pp. 271–277.

[20] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. “JVM Fuzzing for JIT-Induced Side-Channel Detection”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE ’20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1011–1023. DOI: `10.1145/3377811.3380432`. URL: `https://doi.org/10.1145/3377811.3380432`.

[21] Axel Buchner et al. *G*Power 3.1 manual*. Düsseldorf, 2020.

[22] Alexandra Bugariu and Peter Müller. “Automatically Testing String Solvers”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE ’20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1459–1470. DOI: `10.1145/3377811.3380398`. URL: `https://doi.org/10.1145/3377811.3380398`.

[23] *Meta Bug Bounty Program Info*. `https://www.facebook.com/whitehat`. Accessed: November 2022.

[24] *Microsoft Bug Bounty Program*. `https://www.microsoft.com/en-us/msrc/bounty`. Accessed: November 2022.

[25] *Security Bug Bounty Program*. `https://www.mozilla.org/en-US/security/bug-bounty/`. Accessed: November 2022.

[26] *Bugs found in Chrome with fuzzing*. `https://bugs.chromium.org/p/chromium/issues/list?q=label%3AClusterFuzz%20-status%3AWontFix%2CDuplicate&can=1`. Accessed: February 2021.

[27] Benjamin Carlisle et al. “On the other side of the table: Hosting capture the flag (ctf) competitions”. In: *Proceedings of the 6th Workshop on Security Information Workers, ser. WSIW*. Vol. 20. 2020.

[28] *Catalogue Fuzzing*. `https://wcventure.github.io/FuzzingPaper/`. Accessed: June 2022.

[29] *Checkmarx SAST*. `https://www.checkmarx.com/de/products/static-application-security-testing`. Accessed: November 2022.

[30] *Checkmarx SAST license agreement*. `https://checkmarx.atlassian.net/wiki/spaces/CCD/pages/1253442222/CxIAST+End+User+License+Agreement+EULA`. Accessed: November 2022.

[31] Hongxu Chen et al. “MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2325–2342. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu`.

[32] Yaohui Chen et al. “SAVIOR: Towards Bug-Driven Hybrid Testing”. In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1580–1596. DOI: `10.1109/SP40000.2020.00002`. URL: `https://doi.org/10.1109/SP40000.2020.00002`.

[33] Yuanliang Chen et al. “EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers”. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1967–1983. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang`.

[34] B. Chess and G. McGraw. “Static analysis for security”. In: *IEEE Security & Privacy* 2.6 (2004), pp. 76–79. DOI: `10.1109/MSP.2004.111`.

[35] Jaeseung Choi et al. “NtFuzz: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis”. In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 677–693. DOI: `10.1109/SP40001.2021.00114`. URL: `https://doi.org/10.1109/SP40001.2021.00114`.

[36] Maria Christakis and Christian Bird. “What developers want and need from program analysis: an empirical study”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. Ed. by David Lo, Sven Apel, and Sarfraz Khurshid. ACM, 2016, pp. 332–343. DOI: `10.1145/2970276.2970347`. URL: `https://doi.org/10.1145/2970276.2970347`.

[37] *CISCO Fuzzing*. `https://blogs.cisco.com/security/talos/mutiny-decept`. Accessed: June 2022.

[38] clang. *Clang: a C language family frontend for LLVM*. `https://clang.llvm.org/`. Accessed: October 2022.

[39] *Clang Static Analyzer*. `https://clang-analyzer.llvm.org/`. Accessed: November 2022.

[40] *Google Fuzzing*. `https://google.github.io/clusterfuzz/`. Accessed: June 2022.

[41] *CodeSonar SAST*. `https://www.grammatech.com/codesonar-cc`. Accessed: November 2022.

[42] *CodeSonar SAST license agreement*. `https://support.grammatech.com/documentation/licenses/GrammaTech_License_Agreement_CodeSonar_ver.2016.1.0.pdf`. Accessed: November 2022.

[43] Jake Corina et al. “DIFUZE: Interface Aware Fuzzing for Kernel Drivers”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2123–2138. DOI: `10.1145/3133956.3134069`. URL: `https://doi.org/10.1145/3133956.3134069`.

[44] *Coverity Scan*. `https://scan.coverity.com/`. Accessed: November 2022.

[45] *Coverity Scan license agreement*. `https://www.synopsys.com/company/legal/software-integrity/coverity-product-license-agreement.html`. Accessed: November 2022.

[46] Geoff Cumming and Sue Finch. “Inference by eye: confidence intervals and how to read pictures of data.” In: *American psychologist* 60.2 (2005), p. 170.

[47] A. Danilova et al. “Testing Time Limits in Screener Questions for Online Surveys with Programmers”. In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 2080–2090. DOI: `10.1145/3510003.3510223`. URL: `https://doi.ieeecomputersociety.org/10.1145/3510003.3510223`.

[48] Anastasia Danilova et al. “Code Reviewing as Methodology for Online Security Studies with Developers - A Case Study with Freelancers on Password Storage”. In: *Seventeenth Symposium on Usable Privacy and Security, SOUPS 2021, August 8-10, 2021*. Ed. by Sonia Chiasson. USENIX Association, 2021, pp. 397–416. URL: `https://www.usenix.org/conference/soups2021/presentation/danilova`.

[49] Anastasia Danilova et al. “Code Reviewing as Methodology for Online Security Studies with Developers - A Case Study with Freelancers on Password Storage”. In: *Seventeenth Symposium on Usable Privacy and Security, SOUPS 2021, August 8-10, 2021*. Ed. by Sonia Chiasson. USENIX Association, 2021, pp. 397–416. URL: `https://www.usenix.org/conference/soups2021/presentation/danilova`.

[50] Anastasia Danilova et al. “Do You Really Code? Designing and Evaluating Screening Questions for Online Surveys with Programmers”. In: *Proceedings of the 43rd International Conference on Software Engineering*. ICSE ’21. Madrid, Spain: IEEE Press, 2021, pp. 537–548. ISBN: 9781450390859. DOI: `10.1109/ICSE43902.2021.00057`. URL: `https://doi.org/10.1109/ICSE43902.2021.00057`.

[51] Anastasia Danilova et al. “Replication: On the Ecological Validity of Online Security Developer Studies: Exploring Deception in a Password-Storage Study with Freelancers”. In: *Sixteenth Symposium on Usable Privacy and Security, SOUPS 2020, August 7-11, 2020*. Ed. by Heather Richter Lipford and Sonia Chiasson. USENIX Association, 2020, pp. 165–183. URL:

https://www.usenix.org/conference/soups2020/presentation/danilova.

[52] Anastasia Danilova et al. “Replication: On the Ecological Validity of Online Security Developer Studies: Exploring Deception in a Password-Storage Study with Freelancers”. In: *Sixteenth Symposium on Usable Privacy and Security, SOUPS 2020, August 7-11, 2020*. Ed. by Heather Richter Lipford and Sonia Chiasson. USENIX Association, 2020, pp. 165–183. URL: https://www.usenix.org/conference/soups2020/presentation/danilova.

[53] Melissa Dark et al. “Effect of the Secure Programming Clinic on Learners' Secure Programming Practices”. In: *Journal of The Colloquium for Information Systems Security Education*. Vol. 4. 1. 2016, pp. 18–18.

[54] *DARPA Cyber Grand Challenge*. https://www.darpa.mil/program/cyber-grand-challenge. Accessed: November 2022.

[55] A. C. Davison and D. V. Hinkley. *Bootstrap Methods and their Application*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1997. DOI: 10.1017/CBO9780511802843.

[56] *Deutsche Telekom Fuzzing*. https://www.code-intelligence.com/case-study-telekom. Accessed: June 2022.

[57] Sushant Dinesh et al. “RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization”. In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1497–1511. DOI: 10.1109/SP40000.2020.00009. URL: https://doi.org/10.1109/SP40000.2020.00009.

[58] Ren Ding et al. “Hardware Support to Improve Fuzzing Performance and Precision”. In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim et al. ACM, 2021, pp. 2214–2228. DOI: 10.1145/3460120.3484573. URL: https://doi.org/10.1145/3460120.3484573.

[59] Sung Ta Dinh et al. “Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases”. In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. URL: https://www.ndss-symposium.org/ndss-paper/favocado-fuzzing-the-binding-code-of-javascript-engines-using-semantically-correct-test-cases/.

[60] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. “Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations”. In: *IEEE Transactions on Software Engineering* 48.3 (2022), pp. 835–847. DOI: 10.1109/TSE.2020.3004525.

[61] Brendan Dolan-Gavitt et al. “LAVA: Large-Scale Automated Vulnerability Addition”. In: *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016* (2016), pp. 110–121. DOI: 10.1109/SP.2016.15.

[62] Zhengjie Du et al. “WindRanger: A Directed Greybox Fuzzer driven by DeviationBasic Blocks”. In: *Proceedings of the 44th International Conference on Software Engineering (ICSE '22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[63] Paul D Ellis. *The essential guide to effect sizes: Statistical power, meta-analysis, and the interpretation of research results*. Cambridge university press, 2010.

[64] UK Defence Standardization - Defence Equipment and Support. “Requirements for Safety of Programmable Elements (PE) in Defence Systems Part: 01 : Requirements and Guidance”. In: (2021).

[65] Xiaotao Feng et al. “Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference”. In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim et al. ACM, 2021, pp. 337–350. DOI: `10.1145/3460120.3484543`. URL: `https://doi.org/10.1145/3460120.3484543`.

[66] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. “The Use of Likely Invariants as Feedback for Fuzzers”. In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 2829–2846.

[67] Andrea Fioraldi et al. “AFL++ : Combining Incremental Steps of Fuzzing Research”. In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. URL: `https://www.usenix.org/conference/woot20/presentation/fioraldi`.

[68] Paul Fiterau-Brostean et al. “Analysis of DTLS Implementations Using Protocol State Fuzzing”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2523–2540. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean`.

[69] *Frama-C*. `https://frama-c.com/`. Accessed: November 2022.

[70] Kelsey R. Fulton et al. “Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study”. In: *Seventeenth Symposium on Usable Privacy and Security, SOUPS 2021, August 8-10, 2021*. Ed. by Sonia Chiasson. USENIX Association, 2021, pp. 597–616. URL: `https://www.usenix.org/conference/soups2021/presentation/fulton`.

[71] Shuitao Gan et al. “GREYONE: Data Flow Sensitive Fuzzing”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2577–2594. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/gan`.

[72] Xiang Gao et al. "Fuzz Testing Based Data Augmentation to Improve Robustness of Deep Neural Networks". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1147–1158. DOI: 10.1145/3377811.3380415. URL: https://doi.org/10.1145/3377811.3380415.

[73] *Gartner Magic Quadrant for Application Security Testing*. https://www.gartner.com/en/documents/3984345. Accessed: February 2021.

[74] gcc. *GCC, the GNU Compiler Collection*. https://gcc.gnu.org/. Accessed: October 2022.

[75] *GDB web page*. https://www.sourceware.org/gdb/. Accessed: February 2022.

[76] *Go Fuzzing*. https://go.dev/security/fuzz/. Accessed: November 2022.

[77] Google. *ClusterFuzz*. https://google.github.io/clusterfuzz/. Accessed: May 2022.

[78] Google. *Google fuzzing tutorial*. https://github.com/google/fuzzing/blob/master/tutorial/libFuzzerTutorial.md. Accessed: July 2022.

[79] Peter Leo Gorski et al. "Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse". In: *Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018, Baltimore, MD, USA, August 12-14, 2018*. Ed. by Mary Ellen Zurko and Heather Richter Lipford. USENIX Association, 2018, pp. 265–281. URL: https://www.usenix.org/conference/soups2018/presentation/gorski.

[80] Peter Leo Gorski et al. "Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–13. ISBN: 9781450367080. DOI: 10.1145/3313831.3376142. URL: https://doi.org/10.1145/3313831.3376142.

[81] Harrison Green and Thanassis Avgerinos. "GraphFuzz: Library API Fuzzing with Lifetime-aware Dataflow Graphs". In: *Proceedings of the 44th International Conference on Software Engineering (ICSE '22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[82] M. Green and M. Smith. "Developers are Not the Enemy!: The Need for Usable Security APIs". In: *IEEE Security & Privacy* 14.05 (Sept. 2016), pp. 40–46. ISSN: 1558-4046. DOI: 10.1109/MSP.2016.111.

[83] Jiazhen Gu et al. "Muffin: Testing Deep Learning Libraries via Neural Architecture Fuzzing". In: *Proceedings of the 44th International Conference on Software Engineering (ICSE '22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[84] Marco Gutfleisch et al. “How Does Usable Security (Not) End Up in Software Products? Results From a Qualitative Interview Study”. In: *43rd IEEE Symposium on Security and Privacy, IEEE S&P 2022, May 22-26, 2022*. IEEE Computer Society, May 2022.

[85] Marco Gutfleisch et al. “How Does Usable Security (Not) End Up in Software Products? Results From a Qualitative Interview Study”. In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 893–910. DOI: 10.1109/SP46214.2022.9833756. URL: https://doi.org/10.1109/SP46214.2022.9833756.

[86] Yu Hao et al. “Demystifying the Dependency Challenge in Kernel Fuzzing”. In: *Proceedings of the 44th International Conference on Software Engineering (ICSE '22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[87] Xiaoyu He et al. “SoFi: Reflection-Augmented Fuzzing for JavaScript Engines”. In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim et al. ACM, 2021, pp. 2229–2242. DOI: 10.1145/3460120.3484823. URL: https://doi.org/10.1145/3460120.3484823.

[88] *Honggfuzz*. https://github.com/google/honggfuzz. Accessed: November 2022.

[89] Nicolas Huaman et al. “A Large-Scale Interview Study on Information Security in and Attacks against Small and Medium-sized Enterprises”. In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 1235–1252. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/huaman.

[90] Heqing Huang et al. “Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction”. In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1613–1627. DOI: 10.1109/SP40000.2020.00063. URL: https://doi.org/10.1109/SP40000.2020.00063.

[91] Jaewon Hur et al. “DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs”. In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1286–1303. DOI: 10.1109/SP40001.2021.00103. URL: https://doi.org/10.1109/SP40001.2021.00103.

[92] Suhwan Songand Jaewon Hur et al. “R2Z2: Detecting Rendering Regressions in Web Browsers through Differential Fuzz Testing”. In: *Proceedings of the 44th International Conference on Software Engineering (ICSE '22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[93] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. "How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverity Usage". In: *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 2019, pp. 323–333. DOI: 10.1109/ISSRE.2019.00040.

[94] Nasif Imtiaz et al. "Challenges with responding to static analysis tool alerts". In: *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*. Ed. by Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc. IEEE, 2019, pp. 245–249. DOI: 10.1109/MSR.2019.00049. URL: https://doi.org/10.1109/MSR.2019.00049.

[95] *Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts*. Standard. ISO/TC 159/SC 4 Ergonomics of human-system interaction, Mar. 2018.

[96] Kyriakos Ispoglou et al. "FuzzGen: Automatic Fuzzer Generation". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2271–2287. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou.

[97] Bahruz Jabiyev et al. "T-Reqs: HTTP Request Smuggling with Differential Fuzzing". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim et al. ACM, 2021, pp. 1805–1820. DOI: 10.1145/3460120.3485384. URL: https://doi.org/10.1145/3460120.3485384.

[98] Zu-Ming Jiang et al. "Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2595–2612.

[99] Brittany Johnson et al. "Why don't software developers use static analysis tools to find bugs?" In: *Proceedings of the 2013 International Conference on Software Engineering* (2013), pp. 672–681. ISSN: 02705257. DOI: 10.1109/ICSE.2013.6606613. URL: http://dl.acm.org/citation.cfm?id=2486788.2486877.

[100] S. C. Johnson and Murray Hill. "Lint, a C Program Checker". In: 1978.

[101] *JQ*. https://github.com/stedolan/jq. Accessed: November 2022.

[102] Jinho Jung et al. "WINNIE : Fuzzing Windows Applications with Harness Synthesis and Fast Cloning". In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. URL: https://www.ndss-symposium.org/ndss-paper/winnie-fuzzing-windows-applications-with-harness-synthesis-and-fast-cloning/.

[103] Harjot Kaur et al. “Where to Recruit for Security Development Studies: Comparing Six Software Developer Samples”. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 4041–4058. ISBN: 978-1-939133-31-1. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/kaur`.

[104] Hyungsub Kim et al. “PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles”. In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. URL: `https://www.ndss-symposium.org/ndss-paper/pgfuzz-policy-guided-fuzzing-for-robotic-vehicles/`.

[105] Kyungtae Kim et al. “HFL: Hybrid Fuzzing on the Linux Kernel”. In: *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. URL: `https://www.ndss-symposium.org/ndss-paper/hfl-hybrid-fuzzing-on-the-linux-kernel/`.

[106] George Klees et al. “Evaluating Fuzz Testing”. In: CCS ’18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 2123–2138. DOI: `10.1145/3243734.3243804`. URL: `https://doi.org/10.1145/3243734.3243804`.

[107] Katharina Krombholz et al. “"I Have No Idea What I’m Doing" - On the Usability of Deploying HTTPS”. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1339–1356. ISBN: 978-1-931971-40-9. URL: `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/krombholz`.

[108] Jacob Krüger et al. “Do you remember this source code?” In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by Michel Chaudron et al. ACM, 2018, pp. 764–775. DOI: `10.1145/3180155.3180215`. URL: `https://doi.org/10.1145/3180155.3180215`.

[109] James Kukucka et al. “CONFETTI: Amplifying Concolic Guidance for Fuzzers”. In: *Proceedings of the 44th International Conference on Software Engineering (ICSE ’22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[110] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. “Constraint-guided Directed Greybox Fuzzing”. In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 3559–3576. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu`.

[111] Suyoung Lee et al. "Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2613–2630.

[112] Teemu Lehtinen, Aleksi Lukkarinen, and Lassi Haaranen. "Students Struggle to Explain Their Own Program Code". In: *ITiCSE 2021: 26th ACM Conference on Innovation and Technology in Computer Science Education, Virtual Event, Germany, June 26 - July 1, 2021*. Ed. by Carsten Schulte et al. ACM, 2021, pp. 206–212. DOI: `10.1145/3430665.3456322`. URL: `https://doi.org/10.1145/3430665.3456322`.

[113] Teemu Lehtinen, André L. Santos, and Juha Sorva. "Let's Ask Students About Their Programs, Automatically". In: *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 2021, pp. 467–475. DOI: `10.1109/ICPC52881.2021.00054`. URL: `https://doi.org/10.1109/ICPC52881.2021.00054`.

[114] Wenqiang Li et al. "muAFL: Non-intrusive Feedback-driven Fuzzing for Microcontroller Firmware". In: *Proceedings of the 44th International Conference on Software Engineering (ICSE '22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[115] Yuwei Li et al. "UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2777–2794. ISBN: 978-1-939133-24-3. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei`.

[116] Hongliang Liang et al. "Fuzzing: State of the Art". In: *IEEE Trans. Reliab.* 67.3 (2018), pp. 1199–1218. DOI: `10.1109/TR.2018.2834476`. URL: `https://doi.org/10.1109/TR.2018.2834476`.

[117] *libFuzzer*. `https://llvm.org/docs/LibFuzzer.html`. Accessed: November 2022.

[118] *libroxml GitHub*. `https://github.com/blunderer/libroxml`. Accessed: February 2022.

[119] *libroxml web page*. `https://www.libroxml.net/`. Accessed: February 2022.

[120] Daniel Liew et al. "Just Fuzz It: Solving Floating-Point Constraints Using Coverage-Guided Fuzzing". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 521–532. DOI: `10.1145/3338906.3338921`. URL: `https://doi.org/10.1145/3338906.3338921`.

[121] linkedIn. *linkedIn*. `https://www.linkedin.com/`. Accessed: November 2022.

[122] Baozheng Liu et al. “FANS: Fuzzing Android Native System Services via Automated Interface Analysis”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 307–323.

[123] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. “Ankou: Guiding Grey-Box Fuzzing towards Combinatorial Difference”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE ’20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1024–1036. DOI: `10.1145/3377811.3380421`. URL: `https://doi.org/10.1145/3377811.3380421`.

[124] Ruijie Meng et al. “Linear-time Temporal Logic guided Greybox Fuzzing”. In: *Proceedings of the 44th International Conference on Software Engineering (ICSE ’22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[125] *Microfocus-fortify*. `https://www.microfocus.com/de-de/products/static-code-analysis-sast/overview`. Accessed: November 2022.

[126] *Microfocus-fortify license agreement*. `https://www.microfocus.com/media/documentation/micro_focus_end_user_license_agreement.pdf`. Accessed: November 2022.

[127] *Microsoft Fuzzing*. `https://www.microsoft.com/en-us/research/blog/a-brief-introduction-to-fuzzing-and-why-its-an-important-tool-for-developers/`. Accessed: June 2022.

[128] Barton P. Miller, Lars Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Commun. ACM* 33.12 (1990), pp. 32–44. DOI: `10.1145/96267.96279`. URL: `https://doi.org/10.1145/96267.96279`.

[129] AFL - persistent mode. *american fuzzy lop - persistent mode*. `https://github.com/google/AFL/tree/master/llvm_mode`. Accessed: November 2022.

[130] Facundo Molina, Marcelo d’Amorim, and Nazareno Aguirre. “Fuzzing Class Specifications”. In: *Proceedings of the 44th International Conference on Software Engineering (ICSE ’22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[131] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. “Explaining Static Analysis - A Perspective”. In: *34th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASE Workshops 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 29–32. DOI: `10.1109/ASEW.2019.00023`. URL: `https://doi.org/10.1109/ASEW.2019.00023`.

[132] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. “A Large-Scale Study of Usability Criteria Addressed by Static Analysis Tools”. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, pp. 532–543. ISBN: 9781450393799. DOI: 10.1145/3533767.3534374. URL: https://doi.org/10.1145/3533767.3534374.

[133] Stefan Nagy et al. “Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing”. In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 1683–1700. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/nagy.

[134] Stefan Nagy et al. “Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing”. In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim et al. ACM, 2021, pp. 351–365. DOI: 10.1145/3460120.3484787. URL: https://doi.org/10.1145/3460120.3484787.

[135] Alena Naiakshina. “Don’t Blame Developers! Examining a Password-Storage Study Conducted with Students, Freelancers, and Company Developers”. PhD thesis. University of Bonn, Germany, 2020. URL: https://hdl.handle.net/20.500.11811/8842.

[136] Alena Naiakshina et al. “"If You Want, I Can Store the Encrypted Password": A Password-Storage Field Study with Freelance Developers”. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: Association for Computing Machinery, 2019, pp. 1–12. DOI: 10.1145/3290605.3300370. URL: https://doi.org/10.1145/3290605.3300370.

[137] Alena Naiakshina et al. “Deception Task Design in Developer Password Studies: Exploring a Student Sample”. In: *Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018, Baltimore, MD, USA, August 12-14, 2018*. Ed. by Mary Ellen Zurko and Heather Richter Lipford. USENIX Association, 2018, pp. 297–313. URL: https://www.usenix.org/conference/soups2018/presentation/naiakshina.

[138] Alena Naiakshina et al. “On Conducting Security Developer Studies with CS Students: Examining a Password-Storage Study with CS Students, Freelancers, and Company Developers”. In: *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*. Ed. by Regina Bernhaupt et al. ACM, 2020, pp. 1–13. DOI: 10.1145/3313831.3376791. URL: https://doi.org/10.1145/3313831.3376791.

[139] Alena Naiakshina et al. "Why Do Developers Get Password Storage Wrong?: A Qualitative Usability Study". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by Bhavani M. Thuraisingham et al. New York, NY, USA: Association for Computing Machinery, 2017, pp. 311–328. DOI: 10.1145/3133956.3134082. URL: https://doi.org/10.1145/3133956.3134082.

[140] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[141] Hoang Lam Nguyen and Lars Grunske. "BeDivFuzz: Integrating Behavioral Diversity into Generator-based Fuzzing". In: *Proceedings of the 44th International Conference on Software Engineering (ICSE '22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[142] Hoang Lam Nguyen et al. "MoFuzz: A Fuzzer Suite for Testing Model-Driven Software Engineering Tools". In: *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 1103–1115. DOI: 10.1145/3324884.3416668. URL: https://doi.org/10.1145/3324884.3416668.

[143] Tai D. Nguyen et al. "SFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 778–788. DOI: 10.1145/3377811.3380334. URL: https://doi.org/10.1145/3377811.3380334.

[144] Jakob Nielsen. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994.

[145] Yannic Noller et al. "HyDiff: Hybrid Differential Software Analysis". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1273–1285. DOI: 10.1145/3377811.3380363. URL: https://doi.org/10.1145/3377811.3380363.

[146] Timothy Nosco et al. "The Industrial Age of Hacking". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 1129–1146. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/nosco.

[147] *OCLint*. http://oclint.org/. Accessed: November 2022.

[148] Oleksii Oleksenko et al. "SpecFuzz: Bringing Spectre-type vulnerabilities to the surface". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1481–1498. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/oleksenko.

[149] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. "Generating Highly-structured Input Data by Combining Search-based Testing and Grammar-based Fuzzing". In: ().

[150] oss-orig. *OSS-Fuzz*. `https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html`. Accessed: May 2022.

[151] Sebastian Österlund et al. "ParmeSan: Sanitizer-guided Greybox Fuzzing". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2289–2306. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund`.

[152] Gaoning Pan et al. "V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim et al. ACM, 2021, pp. 2197–2213. DOI: `10.1145/3460120.3484811`. URL: `https://doi.org/10.1145/3460120.3484811`.

[153] Soyeon Park et al. "Fuzzing JavaScript Engines with Aspect-preserving Mutation". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1629–1642. DOI: `10.1109/SP40000.2020.00067`. URL: `https://doi.org/10.1109/SP40000.2020.00067`.

[154] Hui Peng and Mathias Payer. "USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2559–2575. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/peng`.

[155] Theofilos Petsios et al. "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2155–2168. DOI: `10.1145/3133956.3134073`. URL: `https://doi.org/10.1145/3133956.3134073`.

[156] Stephan Plöger, Mischa Meier, and Matthew Smith. "A Qualitative Usability Evaluation of the Clang Static Analyzer and libFuzzer with CS Students and CTF Players". In: *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. USENIX Association, Aug. 2021, pp. 553–572. ISBN: 978-1-939133-25-0. URL: `https://www.usenix.org/conference/soups2021/presentation/ploger`.

[157] Stephan Plöger, Mischa Meier, and Matthew Smith. "A Usability Evaluation of AFL and libFuzzer with CS Students". In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, CHI 2023, Hamburg, Germany, April 23-28, 2023*. Ed. by Albrecht Schmidt et al. ACM,

2023, 186:1–186:18. DOI: `10.1145/3544548.3581178`. URL: `https://doi.org/10.1145/3544548.3581178`.

[158] *pwndbg web page*. `https://github.com/pwndbg/pwndbg`. Accessed: February 2022.

[159] *Radamsa*. `https://gitlab.com/akihe/radamsa`. Accessed: November 2022.

[160] Nilo Redini et al. “Diane: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices”. In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 484–500. DOI: `10.1109/SP40001.2021.00066`. URL: `https://doi.org/10.1109/SP40001.2021.00066`.

[161] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. “Automated unit test generation during software development: a controlled experiment and think-aloud observations”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. Ed. by Michal Young and Tao Xie. ACM, 2015, pp. 338–349. DOI: `10.1145/2771783.2771801`. URL: `https://doi.org/10.1145/2771783.2771801`.

[162] Jan Ruge et al. “Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets”. In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 19–36. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/ruge`.

[163] *Rust Fuzz Book*. `https://rust-fuzz.github.io/book/introduction.html`. Accessed: November 2022.

[164] C. Sadowski et al. “Tricorder: Building a Program Analysis Ecosystem”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 598–608. DOI: `10.1109/ICSE.2015.76`.

[165] Jeff Sauro. *A practical guide to the system usability scale: Background, benchmarks & best practices*. Measuring Usability LLC, 2011.

[166] Sergej Schumilo et al. “HYPER-CUBE: High-Dimensional Hypervisor Fuzzing”. In: *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. URL: `https://www.ndss-symposium.org/ndss-paper/hyper-cube-high-dimensional-hypervisor-fuzzing/`.

[167] Sergej Schumilo et al. “Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types”. In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 2597–2614. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo`.

[168] Kostya Serebryany. "OSS-Fuzz - Google's continuous fuzzing service for open source software". In: Vancouver, BC: USENIX Association, Aug. 2017.

[169] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. "Test suites for benchmarks of static analysis tools". In: *2015 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW 2015* November (2016), pp. 12–15. DOI: `10.1109/ISSREW.2015.7392027`.

[170] Bundesamt für Sicherheit in der Informationstechnik. "Leitfaden zur Entwicklung sicherer Webanwendungen. Empfehlungen und Anforderungen an die Auftragnehmer". In: (2013).

[171] Dag Sjøberg et al. "Conducting realistic experiments in software engineering". In: Feb. 2002, pp. 17–26. DOI: `10.1109/ISESE.2002.1166921`.

[172] Justin Smith, Lisa Nguyen Quang Do, and Emerson Murphy-Hill. "Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security". In: *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*. USENIX Association, Aug. 2020, pp. 221–238. URL: `https://www.usenix.org/conference/soups2020/presentation/smith`.

[173] Justin Smith et al. "How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool". In: *IEEE Transactions on Software Engineering* PP (Feb. 2018), pp. 1–1. DOI: `10.1109/TSE.2018.2810116`.

[174] Justin Smith et al. "Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 248–259. DOI: `10.1145/2786805.2786812`. URL: `https://doi.org/10.1145/2786805.2786812`.

[175] *Suricata*. `https://suricata-ids.org/`. Accessed: November 2022.

[176] Mohammad Tahaei and Kami Vaniea. "Recruiting Participants With Programming Skills: A Comparison of Four Crowdsourcing Platforms and a CS Student Mailing List". In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI '22. New Orleans, LA, USA: Association for Computing Machinery, 2022. ISBN: 9781450391573. DOI: `10.1145/3491102.3501957`. URL: `https://doi.org/10.1145/3491102.3501957`.

[177] Mohammad Tahaei et al. "Security Notifications in Static Analysis Tools: Developers' Attitudes, Comprehension, and Ability to Act on Them". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1–17. DOI: `10.1145/3411764.3445616`. URL: `https://doi.org/10.1145/3411764.3445616`.

[178] *Tesseract OCR*. `https://github.com/tesseract-ocr/tesseract`. Accessed: November 2022.

[179] David R. Thomas. "A General Inductive Approach for Analyzing Qualitative Evaluation Data". In: *American Journal of Evaluation* (), pp. 237–246.

[180] Tyler Thomas et al. "What Questions Remain? An Examination of How Developers Understand an Interactive Static Analysis Tool". In: *2nd Workshop on Security Information Workers, WSIW@SOUPS 2016, Denver, CO, USA, June 22, 2016*. USENIX Association, 2016. URL: `https://www.usenix.org/conference/soups2016/workshop-program/wsiw16/presentation/thomas`.

[181] Christian Tiefenau et al. "A Usability Evaluation of Let's Encrypt and Certbot: Usable Security Done Right". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 1971–1988.

[182] *tomlc99 GitHub*. `https://github.com/cktan/tomlc99`. Accessed: February 2022.

[183] Carmine Vassallo et al. "Context is king: The developer perspective on the usage of static analysis tools". In: *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. Ed. by Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd. IEEE Computer Society, 2018, pp. 38–49. DOI: `10.1109/SANER.2018.8330195`. URL: `https://doi.org/10.1109/SANER.2018.8330195`.

[184] *Veracode SAST*. `https://www.veracode.com/products/binary-static-analysis-sast`. Accessed: November 2022.

[185] Daniel Votipka, Desiree Abrokwa, and Michelle L. Mazurek. "Building and Validating a Scale for Secure Software Development Self-Efficacy". In: CHI '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–20. ISBN: 9781450367080. DOI: `10.1145/3313831.3376754`. URL: `https://doi.org/10.1145/3313831.3376754`.

[186] Daniel Votipka et al. "Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 109–126. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-understanding`.

[187] Haijun Wang et al. "Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 999–1010. DOI: `10.1145/3377811.3380386`. URL: `https://doi.org/10.1145/3377811.3380386`.

[188] Jinghan Wang, Chengyu Song, and Heng Yin. "Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing". In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. URL: `https://www.ndss-symposium.org/ndss-paper/reinforcement-learning-based-hierarchical-seed-scheduling-for-greybox-fuzzing/`.

[189] Yanhao Wang et al. "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization". In: *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. URL: `https://www.ndss-symposium.org/ndss-paper/not-all-coverage-measurements-are-equal-fuzzing-by-coverage-accounting-for-input-prioritization/`.

[190] Anjiang Wei et al. "Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source". In: *Proceedings of the 44th International Conference on Software Engineering (ICSE '22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[191] Charles Weir, Ben Hermann, and Sascha Fahl. "From Needs to Actions to Secure Apps? The Effect of Requirements and Developer Practices on App Security". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 289–305. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/weir`.

[192] Cheng Wen et al. "MemLock: Memory Usage Guided Fuzzing". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 765–777. DOI: `10.1145/3377811.3380396`. URL: `https://doi.org/10.1145/3377811.3380396`.

[193] Dominik Wermke et al. "Committed to Trust: A Qualitative Study on Security & Trust in Open Source Software Projects". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 1880–1896. DOI: `10.1109/SP46214.2022.9833686`.

[194] Alan Woodlief, Sebastian Elbaum, and Kevin Sullivan. "Semantic Image Fuzzing of AI Perception Systems". In: *Proceedings of the 44th International Conference on Software Engineering (ICSE '22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[195] Mingyuan Wu et al. "Evaluating and Improving Neural Program-Smoothing-based Fuzzing". In: *Proceedings of the 44th International Conference on Software Engineering (ICSE '22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[196] Mingyuan Wu et al. "One Fuzzing Strategy to Rule Them All". In: *Proceedings of the 44th International Conference on Software Engineering (ICSE '22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[197] Valentin Wüstholz and Maria Christakis. "Targeted Greybox Fuzzing with Static Lookahead Analysis". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 789–800. DOI: `10.1145/3377811.3380388`. URL: `https://doi.org/10.1145/3377811.3380388`.

[198] Meng Xu et al. "Krace: Data Race Fuzzing for Kernel File Systems". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1643–1660. DOI: `10.1109/SP40000.2020.00078`. URL: `https://doi.org/10.1109/SP40000.2020.00078`.

[199] Wen Xu, Soyeon Park, and Taesoo Kim. "FREEDOM: Engineering a State-of-the-Art DOM Fuzzer". In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. Ed. by Jay Ligatti et al. ACM, 2020, pp. 971–986. DOI: `10.1145/3372297.3423340`. URL: `https://doi.org/10.1145/3372297.3423340`.

[200] Aiko Fallas Yamashita and Leon Moonen. "Do developers care about code smells? An exploratory survey". In: *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*. Ed. by Ralf Lämmel, Rocco Oliveto, and Romain Robbes. IEEE Computer Society, 2013, pp. 242–251. DOI: `10.1109/WCRE.2013.6671299`. URL: `https://doi.org/10.1109/WCRE.2013.6671299`.

[201] Aiko Fallas Yamashita and Leon Moonen. "Surveying developer knowledge and interest in code smells through online freelance marketplaces". In: *2nd International Workshop on User Evaluations for Software Engineering Researchers, USER 2013, San Francisco, CA, USA, May 26, 2013*. IEEE Computer Society, 2013, pp. 5–8. DOI: `10.1109/USER.2013.6603077`. URL: `https://doi.org/10.1109/USER.2013.6603077`.

[202] *yaml-cpp*. `https://github.com/jbeder/yaml-cpp`. Accessed: November 2022.

[203] Tai Yue et al. "EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2307–2324. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/yue`.

[204] Kunpeng Zhang et al. "Path Transitions Tell More: Optimizing Fuzzing Schedules via Runtime Program States". In: *Proceedings of the 44th International Conference on Software Engineering (ICSE '22), May 22–27, 2022, Pittsburgh, PA, USA*. ACM, May 2022.

[205] Zhuo Zhang et al. "StochFuzz: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 659–676. DOI: `10.1109/SP40001.2021.00109`. URL: `https://doi.org/10.1109/SP40001.2021.00109`.

[206] Rui Zhong et al. "SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback". In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. Ed. by Jay Ligatti et al. ACM, 2020, pp. 955–970. DOI: `10.1145/3372297.3417260`. URL: `https://doi.org/10.1145/3372297.3417260`.

[207] Chijin Zhou et al. "Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling". In: *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 858–870. DOI: `10.1145/3324884.3416572`. URL: `https://doi.org/10.1145/3324884.3416572`.

[208] Xiaogang Zhu and Marcel Böhme. "Regression Greybox Fuzzing". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim et al. ACM, 2021, pp. 2169–2182. DOI: `10.1145/3460120.3484596`. URL: `https://doi.org/10.1145/3460120.3484596`.

[209] Peiyuan Zong et al. "FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2255–2269. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/zong`.

[210] zzuf. *zzuf*. `https://github.com/samhocevar/zzuf`. Accessed: November 2022.

Appendix A

Static vs. Dynamic

A.1 Semi-Structured Interview

A.1.1 CS Study

Task 1

- Please explain what you did in the first task.
    - Do you have a point where you want to elaborate on?
    - Did you encounter any problems?
    - Did anything went exceptionally well?
    - Please elaborate on the output of the tool.
    - Can you tell me something about the usability?
    - Where do you see potential for improvement?

Task 2

- Please explain what you did in the second task.
    - Do you have a point where you want to elaborate on?
    - Did you encounter any problems?
    - Did anything went exceptionally well?
    - Please elaborate on the output of the tool.
    - Can you tell me something about the usability?
    - Where do you see potential for improvement?

Comparison

- Please compare the two tasks.
- Do you have anything particular in mind that was comparably easy or hard?
- Would you want to use one of the tools, both or none in the future? Why?

A.1.2 CTF Study

Static

- Please explain what you did in the task.
- How would you rate the usability of the Clang Static Analyzer on a scale from 1-7, 1 very low, 7 very high?
- Please elaborate on the Usability of the Clang Static Analyzer.
- Can you tell me something about the Output of the analyzer?
- What was your biggest problem?
- How would you rate the documentation again on scale from 1-7?

Dynamic

- Please explain what you did in the task.
- How would you rate the usability of libFuzzer on a scale from 1-7, 1 very low, 7 very high?
- Please elaborate on the Usability of libFuzzer.
- Please elaborate on your fuzz target.
- Have you used a dictionary or corpus?
- What did you think of the output?
- How did you interact with the output?
- How did you determine that the fuzzer is running well?

Comparison

- Please compare the two tasks.
- Do you have anything particular in mind that was comparably easy or hard?
- Would you want to use one of the tools, both or none in the future? Why?

general

- What is a security related bug?

A.2 Clang Static Analyzer Overview

Program	**Clang Static Analyzer reports**
Tesseract	476
protobuf 3.9.x	92
protobuf 3.8.x	121
util-linux	142
simple-obfs	15
cmatrix	3
vlc	219
wine	4746
netdata	32
darknet	73
libnice	3
obs-studio	456
jq	4
FFmpeg	639
yuzu	339
spdlog	0
simdjson	2

TABLE A.1: Overview of GitHub projects and reports of Clang Static Analyzer

A.3 Overview of Task Ordering

first	started	drop-out	submitted	success
Static-easy	8	1	7	6
Static-hard	7	4	3	0
second	started	drop-out	submitted	success
Static-easy	4	2	2	2
Static-hard	3	1	2	0
combined	started	drop-out	submitted	success
Static-easy	12	3	9	8
Static-hard	10	5	5	0

TABLE A.2: CS: static analysis overall statistics

first	started	drop-out	submitted	success
Fuzzing-easy	9	5	4	1
Fuzzing-hard	7	4	3	0
second	started	drop-out	submitted	success
Fuzzing-easy	7	3	4	1
Fuzzing-hard	3	2	1	0
combined	started	drop-out	submitted	success
Fuzzing-easy	16	8	8	2
Fuzzing-hard	10	6	4	0

TABLE A.3: CS: fuzzing overall statistics

first	started	drop-out	submitted	success
Fuzzing	3	0	3	0
Static	3	0	3	0
second	started	drop-out	submitted	success
Fuzzing	3	0	3	0
Static	3	0	3	1
combined	started	drop-out	submitted	success
Fuzzing	6	0	6	0
Static	6	0	6	1

TABLE A.4: CTF: static analysis and fuzzing overall statistics

Participant	Static				Dynamic								
	Step 1	Step 2	Step 3	Bug	Step 1	Step 2	Step 3	Step 4	Step 5	Bug	Step 6	Corpus	Toy
CS27-SFE	no submission				not started								
CS31-SFE	✗	✓	✓	✓	✗ / ✗								✓
CS1-SFE	✓	✓	✓	✓	no submission								
CS9-SFE	✓	✓	✓	✓	no submission								
CS19-SFE	✓	✓	✓	✓	no submission								
CS15-SFE	✓	✓	✓	✓	2 / ✓	2	✗ / ✗ (FT in TPr)	✗					✗
CS5-SFE	✓	✓	✓	✓	1 / ✓	1	✓ / ✗	✓	✓	✓	✗	✓	✗
CS23-SFE	✓	✓	✓	✓	1 / ✓	1	✓ / ✓	✓	✓	✓	✓	✓	✗
CS21-SFH	no submission				not started								
CS25-SFH	no submission				not started								
CS29-SFH	no submission				not started								
CS7-SFH	✓	✓	✓	✗	no submission								
CS13-SFH	✓	✓	✓	✗	no submission								
CS17-SFH	✓	✓	✓	✗	✗ / ✗								✗
CS3-SFH	✓	✓	✓	✗	❍ / ❍	❍	✗ / ✗	✗					✓

Participant	Dynamic									Static			
	Step 1	Step 2	Step 3	Step 4	Step 5	Bug	Step 6	Corpus	Toy	Step 1	Step 2	Step 3	Bug
CS2-FSE	no submission									not started			
CS10-FSE	no submission									not started			
CS12-FSE	no submission									not started			
CS20-FSE	no submission									not started			
CS32-FSE	no submission									not started			
CS11-FSH	no submission									not started			
CS14-FSH	no submission									not started			
CS22-FSH	no submission									not started			
CS28-FSE	✗ / ✗									no submission			
CS16-FSE	✗ / ✗								✗	✓	✓	✓	✓
CS24-FSE	1 / ✓	1	✓ / ✗	✗					✓	✓	✓	✓	✓
CS6-FSE	2 / ✓	2	✓ / ✗	✓	✓	✓	✓	✗	✗	no submission			
CS18-FSH	✗ / ✗								✗	✗			
CS26-FSH	❍ / ❍	❍		❍ / ❍					✓	✓	✓	✓	✗
CS4-FSH	✗ / ✗								✓	✓	✓	✓	✗
CS8-FSH	1 / ✗	❍	✗ / ✗						✓	✓	✓	✓	✗
CS30-FSH	✗ / ✗		✗ / ❍							not started			

TABLE A.5: CS overall

Participant	Static				Dynamic								
	Step 1	Step 2	Step 3	Bug	Step 1	Step 2	Step 3	Step 4	Step 5	Bug	Step 6	Corpus	Toy
CTF1-SF	✓	✓	✓	✗	no submission								
CTF5-SF	✓	✓	✓	✗	1 / ✓	1	✗ / ✗	✗					✗
CTF3-SF	✓	✓	✓	✗	1 / ✓	3+	✓ / ✓ (FT in TPr)	✓	✓	✗	✓	✓	✗
CTF7-SF	✓	✓	✓	✗	1 / ✓	11	✓ / ✓	✓	✓	✗	✓	✓	✗

Participant	Dynamic									Static			
	Step 1	Step 2	Step 3	Step 4	Step 5	Bug	Step 6	Corpus	Toy	Step 1	Step 2	Step 3	Bug
CTF8-FS	no submission									not started			
CTF4-FS	1 / ✓	1	✓ / ✓ (FT in TPr)	❍	❍	✗	✓		✗	✓	✓	✓	✗
CTF2-FS	1 / ✓	1	✓ / ✓ (FT in TPr)	✓	✓	✗	✓	✓	✗	✓	✓	✓	✗
CTF6-FS	1 / ✓	10	✓ / ✓ (FT in TPr)	✓	✓	✗	❍	✗	✗	✓	✓	✓	✓

TABLE A.6: CTF overall

A.4 Comments and Usage in Future

Participant	Comment				Use in Future		
	static		dynamic				
	positive	negative	positive	negative	static	dynamic	none
CS5-SFE	4	2	2	9	✓	✓	
CS15-SFE	4	4	1	7	✓	✓	
CS16-FSE	6	1	0	5	✓		
CS23-SFE	3	3	1	9	✓		
CS24-FSE	8	7	3	3	❍	❍	❍
CS31-FSE	2	4	1	2			✓
∑ easy	27	21	8	35	4	2	1
CS3-SFH	8	6	0	5	✓		
CS4-FSH	6	6	2	6	✓	✓	
CS8-FSH	6	6	0	7	✓		
∑ hard	20	18	2	18	3	1	0
∑	47	39	10	53	7	3	1

TABLE A.7: CS: Comments and usage in future of the static and dynamic analysis tools

Participant	Comment				Use in Future		
	static		dynamic				
	positive	negative	positive	negative	static	dynamic	none
CTF2-FS	2	3	2	10	✓	✓	
CTF3-SF	2	5	1	2	✓	✓	
CTF4-FS	2	5	1	6		✓	
CTF5-SF	4	2	0	4	✓	✓	
CTF6-FS	2	4	2	5	clean code	✓	
CTF7-SF	8	5	0	4		✓	
∑	20	24	6	31	3	6	0

TABLE A.8: CTF Comments and usage in future of the static and dynamic analysis tools

A.5 Pre-Questionnaire

First the pre-questionnaire for the CS participants is presented, followed by the pre-questionnaire for the CTF participants.

A.5.1 Pre-Questionnaire CS Students

- How many years of programming experience do you have?
  [Number]

- Which programming languages are you proficient in?
  Java, C, C++, Python, Ruby, Go, Pascal, Basic, Rust, Perl, PHP, JavaScript, R, Other (please specify) [Free text]
- How proficient are you with the following operating systems?
  1 - not at all - 7 - excellent
  Windows, Linux, Mac, BSD
- How many hours per week do you spend programming?
  [Number]
- Have you ever used static code analysis to support your programming?
  Yes, No
- *if yes*
  - Which static code analysis tools have you ever used?
    [Free text]
  - Please rate your agreement with the following statement: static code analysis supported my programming!
    1 - Disagree very strongly - 7 - agree very strongly
- Are your familiar with the term 'fuzzing'?
  Yes, No
- Have you ever used a fuzzer?
  Yes, No
- *if yes*
  - Please name all fuzzers you have ever used.
    [Free text]
  - Have you ever found a bug with a fuzzer?
    Yes, No
- Please rate how proficient you are in the following programming languages.
  1 - not at all - 7 - excellent
  Java, C, C++, Python, Ruby, Go, Pascal, Basic, Rust, Perl, PHP, JavaScript, R
- Please state your age
  [Number]
- Please state your gender
  male, female, other, no answer

A.5.2 Pre-Questionnaire CTF Players

The CTF players were asked the same question in the pre-questionnaire as the CS students shown in Section A.5.1. Additionally, they were also asked the following question:

- What is your main occupation?
  Student, Employee, Employer, Freelancer, job-seeking, no answer, Other (please specify) [Free text]
- *if Employee was selected*
  - How many employees does your company have?
    [Number]
  - How many members on average does the team you are working in have?
    [Number]
- *if Student was selected*
  - What is your intended level of education?
    [Free text]
  - What is your major at the moment?
    [Free text]
  - What is your current semester?
    [Number]
- Please state your level of education.
  Master of Computer Science or comparable, Bachelor of Computer Science or comparable, Ausbildung Fachinformatiker or comparable, Abitur, High school diploma, Mittlere Reife, no answer, Other (please specify) [Free text]

Appendix B

AFL vs. libFuzzer

B.1 Pre-Questionnaire

Programming Experience

- How many years of programming experience do you have?
  [Number]
- Which programming languages are you proficient in?
  Java, C, C++, Python, Ruby, Go, Pascal, Basic, Rust, Perl, PHP, JavaScript, R, Other (please specify) [Free text]
- Please rate how proficient you are in the following programming languages.
  1 - not at all - 7 - excellent
  Java, C, C++, Python, Ruby, Go, Pascal, Basic, Rust, Perl, PHP, JavaScript, R

Static Code Analysis

- Have you ever used static code analysis to support your programming?
  Yes, No
- *if yes*
  - Which static code analysis tools have you ever used?
    [Free text]
  - Please rate your agreement with the following statement: static code analysis supported my programming!
    1 - Disagree very strongly - 7 - agree very strongly

Dynamic Code Analysis

- Are your familiar with the term ’fuzzing’?
  Yes, No

- Have you ever used a fuzzer?
  Yes, No
- *if yes*
  - Please name all fuzzers you have ever used.
    [Free text]
  - Have you ever found a bug with a fuzzer?
    Yes, No

Demographics

- Please select your gender.
  Male, Female, Prefer not say, Prefer to self-describe: [Free text
- Please enter your age:
  [Number]
- How many hours per week do you spend programming?
  [Number]

B.2 How to Solve the Tasks

In this section, we present possible solutions for our tasks. In order to save space, we provide a solution for each target program with only one fuzzer.

B.2.1 AFL and tomlc99

In order to fuzz tomlc99, we choose the toml_parse function.

The fuzz target with the signature while-loop for using the persistent mode is shown in Listing B.1. It reads the input from stdin, writes the input into a buffer, and calls toml_parse with this buffer. Moreover, an error buffer is created and passed to the function. In the end, the resulting toml table is freed.

The target program is built with the command presented in Listin B.2. The make command is invoked with the compiler set to the AFL compiler. Then the fuzz target is built with the command also shown in B.2. To do so, the AFL compiler is used, the include path is extended to the path to the build folder, and the library to link against is specified. In both compilations, the variable AFL_USE_ASAN is set to 1 to use the address sanitizer.

Afterward, the fuzzer is run by specifying the input and output folder and the target. The command is shown in Listing B.3.

```
#include <unistd.h>
#include <string.h>
```

```
#include "toml.h"


int main() {
  char buf[100];

  while (__AFL_LOOP(1000)) {
    char errbuf[200];

    memset(buf, 0, 100);
    read(0, buf, 100);

    toml_table_t* tab = toml_parse(buf, errbuf, 200);

    toml_free(tab);
  }

  return 0;
}
```

LISTING B.1: Fuzz Target of AFL for tomlc99

```
# Building Target Program
AFL_USE_ASAN=1 make CC=afl-clang-fast


# Building Fuzz Target
AFL_USE_ASAN=1 afl-clang-fast target.c -I../ \\
    ../libtoml.a -o target
```

LISTING B.2: Build commands of AFL for tomlc99

```
#Running AFL
afl-fuzz -i in -o out -- ./target
```

LISTING B.3: Run commands of AFL for tomlc99

B.2.2 libFuzzer and libroxml

To fuzz libroxml, we choose the function roxml_load_buf.

The fuzz target, which can be seen in Listing B.4, consists of the needed LLVMFuzzerTestOneInput-function. The input argument is written into a buffer and null-terminated. The buffer is then passed to the target function. In the end, the resulting node is closed.

The target program is built with the make command, the compiler is set to clang, and the fuzzer-no-link flag is also passed to instrument the target program. The fuzz target is also built with the clang compiler. The built command for the fuzz target and target program is shown in Listing B.5. Moreover, the fuzzer flag is used for instrumentation and linking the fuzzer. Besides that, the include-path is extended to the path to the build folder, and the library to link against is specified.

The fuzzer is started by running the executable, which is shown in Listing B.6.

```
#include <stddef.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

#include "roxml.h"


int LLVMFuzzerTestOneInput(uint8_t *input, size_t size) {
  char* buf = (char*) malloc(size +1);
  memcpy(buf, input, size);
  buf[size]=0;

  ROXML_API node_t *root = roxml_load_buf(buf);
  roxml_close(root);

  return 0;
}
```

LISTING B.4: Fuzz Target of libFuzzer for libroxml

```
# Building Target Program
./configure CC=clang CFLAGS="-fsanitize=fuzzer-no-link"
make


# Building Fuzz Target
clang target.c -fsanitize=fuzzer -I../src ../.libs/libroxml.a -o
    target
```

LISTING B.5: Build commands of libFuzzer for libroxml

```
#Running libFuzzer
./target
```

LISTING B.6: Run commands of libFuzzer for libroxml

B.3 Semi-Structured Interview

Here we present the questions asked in the final semi-structured Interview.

B.3.1 AFL and libFuzzer

Subtask 1: Get a first impression of the target program and getan overall idea of what the program does.

- What did you do in this subtask?
- What was the most difficult part of this subtask?

Subtask 2: Find a suitable function to fuzz.

- What did you do in this subtask?
- What made you think that the function(s) were suitable?
- What was the most difficult part of this subtask?

Subtask 3: Write your fuzz target in an external file.

- What did you do in this subtask?
- Why do you think should the fuzz target be written in an external file?
- Please elaborate on your fuzz target.
- What was the most difficult part of this subtask?

Subtask 4: Compile and instrument the target program.

- What did you do in this subtask?
- What is meant with target program?
- What is the meaning of instrumentation?
- What was the most difficult part of this subtask?

Subtask 5: Compile the fuzz target.

- What did you do in this subtask?
- What is the connection between subtask 3, 4 and 5?
- What was the most difficult part of this subtask?

Subtask 6: Run the fuzzer and interpret the output.

- What did you do in this subtask?
- How did you interact with the output?
- How did you decide whether the fuzzing is doing something meaningful?
- What was the most difficult part of this subtask?

Subtask 7: If necessary, adjust and improve.

- What did you do in this subtask?
- What was the most difficult part of this subtask?

Conclusion

- Please elaborate on the usability of the fuzzer.
- What was the most difficult subtask in this task?
- What was your biggest problem in this task?
- Do you have any further comments on this task?
- Do you have any further comments regarding the fuzzer?

B.3.2 Comparison

- Please compare both fuzzers.
- Would you use one of the fuzzers, both or none in the future?

B.4 Overview of All Participants

Table B.1 and Table B.2 show the detailed results for all steps of all of our participants that made a submission. Table B.1 shows the results of the condition participants were assigned to first and Table B.2 those for the second assignment. The tables are sorted by fuzzer and target program. Participants highlighted in gray did not manage to complete the condition. Participants highlighted in purple completed the condition but did not complete the study (i.e., both conditions). All participants with a white background completed both conditions.

In the tables the color green denotes whether a participant succeeded on their own and received a point for the step. The color yellow denotes a step a participant succeeded in after receiving support and thus did not get a point. If a participant succeeded in a step after support was given in a step before, it is

Participant	Fuzzer	Program	Sel. Func.	Working FT	Built TPr	Instr. TPr	Built FT	Ran Fuzzer	ASAN in TPr	Corpus/Dict	Trig. UEIC	Trig. Bug	Success	Score
P10	AFL	tomlc99	1 / ✓	1 / ✓	✓	✓	✓	✓	✗*	✓	✗	✗	-	7
P46	AFL	tomlc99	1 / ✓	1 / ✓	✓	✓	✓	✓	✗*	✓	✗	✗	-	7
P26	AFL	tomlc99	✗ ./unitt/t1*	1 / ✗*	✓FTL fuzzing			✗	✗	✗	✗	✗	-	3
P06	AFL	tomlc99	✓ ./toml_cat (FTL fuzzing)					✓	✗*	✓	✗	✗	-	7
P14	AFL	tomlc99	✓ ./toml_json (FTL fuzzing)					✓	✓*	✓	✗	✓	✓	7
P02	AFL	tomlc99	1 / ✓	1 / ✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	10
P49	AFL	libroxml	1 / ✓	1 / ✓	✓	✗	✗	✗	✗	✗	✗	-	-	3
P09	AFL	libroxml	✗ FTL fuzzing*		✓	✗*	✓+	✗+	✗	✗	-	-	-	1
P29	AFL	libroxml	1 / ✗	1 / ✗*	✓	✓	✗	✗	✗	✗	-	-	-	2
P37	AFL	libroxml	2 / ✓	2 / ✓	✓ direct compilation			✓	✗	✓	✗	✓	✓	9
P25	AFL	libroxml	7 / ✓	1 / ✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	9
P33	AFL	libroxml	2 / ✓	2 / ✓	✓ direct compilation			✓	✗	✓	✗	✓	✗	8
P05	AFL	libroxml	1 / ✓	1 / ✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	9
P21	AFL	libroxml	1 / ✓	1 / ✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	10
P41	AFL	libroxml	1 / ✓	1 / ✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	9
P45	AFL	libroxml	1 / ✓	1 / ✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	9
P36	libFuzzer	tomlc99	1 / ✗	1 / ✗*	✓	✗	✗*	-	✗	✗	-	-	-	1
P20	libFuzzer	tomlc99	1 / ✓	1 / ✓	✓	✓	✓	✓	✓	✗	✗	✓	✗	8
P40	libFuzzer	tomlc99	1 / ✗	1 / ✗	✓	✗	✗*	-	✗	✗	-	-	-	1
P04	libFuzzer	tomlc99	1 / ✓	1 / ✓	✓ direct compilation			✓	✓	✓	✗	✓	✓	10
P32	libFuzzer	tomlc99	1 / ✓	1 / ✓	✓ direct compilation			✓	✓	✓	✗	✓	✓	10
P08	libFuzzer	tomlc99	1 / ✓	1 / ✗	✓	✗*	✓	✓	✓	✓	✓!	✗	-	6
P44	libFuzzer	tomlc99	1 / ✓	1 / ✗	✓	✗**	✓	✓	✗	✗	✓	✗	-	4
P12	libFuzzer	tomlc99	1 / ✓	1 / ✓	✓	✗*	✓	✓	✓*	✓	✓	✓	✓	6
P48	libFuzzer	tomlc99	2 / ✓	2 / ✗	✓ direct compilation			✓	✓	✗	✓!	✗	-	6
P16	libFuzzer	tomlc99	2 / ✓	2 / ✓	✓ direct compilation			✓	✓*	✓	✓	✗	-	7
P47	libFuzzer	libroxml	0 / ✗	0 / ✗	✓	✗*	✗+	-	✗	✗	-	-	-	1
P39	libFuzzer	libroxml	10 / ✓	1 / ✗	✓	✗*	✓*	✓	✓	✗	✓	✗	-	3
P23	libFuzzer	libroxml	3 / ✗	3 / ✗	✓	✗	✗*	-	✗	✗	-	-	-	1
P19	libFuzzer	libroxml	1 / ✓	1 / ✗*	✓	✓	✓	✓	✗**	✓	✓	✗	-	6
P15	libFuzzer	libroxml	1 / ✓	1 / ✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	9

TABLE B.1: Results of All Participants for First Task: ✓: success - ✗: failure - green: success without support - yellow: success with support - blue: success after support in previous step - red: no success - gray: not in score

highlighted in blue. This is not awarded with a point since the score should only reflects what a participant was able to reach without any help. The color red denotes a step in which the participant did not succeed and thus also did not receive a point. The column in gray is not part of the score. Participants marked in gray did not finish the task, so only partial information is present, while participants in violet finished the task but dropped out in the next.

If a hint was given in one of the steps, it is denoted with an asterisk, for help the plus sign is used. If a participant did not work on a subtask, e.g., did not investigate whether a crash was a bug and was asked to do so by the study assistant, it is denoted with an exclamation mark. If a step was not reached it is denoted by a dash.

Sel. Func. The number in the column Selected Function shows how many functions the participant tried to fuzz. The tick denotes if the bug could be triggered via one of those function.

Working FT The number in the column Working FuzzTarget depicts the number of fuzz targets a participant wrote. The tick denotes whether a fuzz target could in theory trigger the bug.

Built TPr & Instr. TPr The column Built TargetProgram contains the information on whether the target program was built, while the column Instrumented TargetProgram shows if the target program was instrumented.

Participant	Fuzzer	Program	Sel. Func.	Working FT	Built TPr	Instr. TPr	Built FT	Ran Fuzzer	ASAN in TPr	Corpus/Dict	Trig. UEIC	Trig. Bug	Success	Score
P23	AFL	tomlc99	0 / ✗	0 / ✗*	✓	✗	✗	-	✗	✗	-	-	-	1
P19	AFL	tomlc99	6 / ✓	2 / ✓	✓	✓	✓	✓	✓*	✓	✓	✓	✓	7
P15	AFL	tomlc99	✓ ./toml_cat (FTL fuzzing)				✓	✓	✗	✓	✗	✓	✓	9
P32	AFL	libroxml	1 / ✓	1 / ✓	✓	✓	✓	✓	✗*	✓	✗	✗	✗	7
P48	AFL	libroxml	✓ ./roxml (FTL fuzzing)				✓	✗	✓	✓	-	-	-	7
P08	AFL	libroxml	✓ ./roxml (FTL fuzzing)				✓	✓	✗	✓	✗	✓	✗	8
P44	AFL	libroxml	✓ ./rocat (FTL fuzzing)				✓	✓	✗	✓	✗	✓	✗!	8
P12	AFL	libroxml	✓ ./rocat (FTL fuzzing)				✓	✓	✓	✓	✗	✓	✓	10
P16	AFL	libroxml	1 / ✓	1 / ✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	10
P29	libFuzzer	tomlc99	1 / ✓	1 / ✗	✓	✗*	✓	✓	✗	✗	✗	✗	-	4
P37	libFuzzer	tomlc99	3 / ✓	3 / ✓	✓	✗*	✓	✓	✗*	✓	✓	✗	-	6
P21	libFuzzer	tomlc99	10 / ✓	6 / ✗	✓direct compilation			✓	✓	✓	✓	✗	-	7
P25	libFuzzer	tomlc99	4 / ✓	1 / ✗	✓direct compilation			✓	✓*	✓	✓	✗	-	6
P33	libFuzzer	tomlc99	2 / ✓	2 / ✓	✓	✗	✓	✓	✗	✓	✓	✗	-	6
P45	libFuzzer	tomlc99	5 / ✓	1 / ✓	✓	✗*	✓	✓	✗*	✓	✓	✗	-	6
P41	libFuzzer	tomlc99	1 / ✓	1 / ✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	8
P05	libFuzzer	tomlc99	3 / ✓	3 / ✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	9
P26	libFuzzer	libroxml	3 / ✗	2 / ✗*	✓	✗	✓	✓	✗	✗	✗	-	-	3
P06	libFuzzer	libroxml	2 / ✓	1 / ✗	✓	✗*	✓	✓	✗*	✓	✓	✗	-	5
P02	libFuzzer	libroxml	1 / ✓	1 / ✓	✓direct compilation			✓	✓	✗	✗	✓	✓	9
P14	libFuzzer	libroxml	1 / ✓	1 / ✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	10

TABLE B.2: Results of All Participants for Second Task: ✓: success - ✗: failure - green: success without support - yellow: success with support - violet: success after support in previous step - red: no success - gray: not in score

Built FT The column Built FuzzTarget shows whether the fuzz target was built correctly.

Ran Fuzzer The column Ran Fuzzer shows if the fuzzer was run.

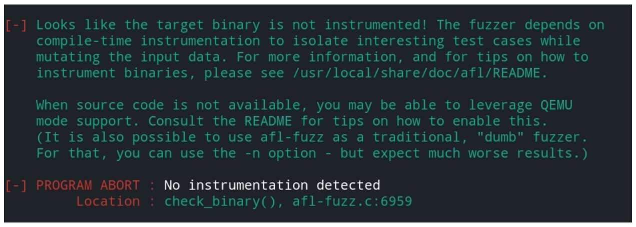
ASAN in TPr The column ASAN in TargetProgram shows whether the target program was built with the address sanitizer. The address sanitizer is of interest since it is needed to find the known bug in tomlc99. We awarded points in the libroxml condition as well since participants could not know that it was not needed, and it also helps when interpreting crashes.

Corpus/Dict The column Corpus/Dictionary shows whether participants used a corpus or dictionary. While this was not necessary to find the bugs, which the participants did not know, it is a common and recommendable optimization.

Trig. UEIC The column Triggered UEIC (User Error Induced Crash) shows whether a crash was induced by an error of the user, and the column Triggered Bug shows whether an actual bug in the target program was triggered.

Success The column Success shows if participants correctly interpreted the output as a bug thus completing the overall task. This column also matches the success score of Plöger et al.'s study.

Score The fuzzing score is shown in the second to last column and is the sum off all subtasks which participants successfully solved without support.



```
[-] Looks like the target binary is not instrumented! The fuzzer depends on
    compile-time instrumentation to isolate interesting test cases while
    mutating the input data. For more information, and for tips on how to
    instrument binaries, please see /usr/local/share/doc/afl/README.

    When source code is not available, you may be able to leverage QEMU
    mode support. Consult the README for tips on how to enable this.
    (It is also possible to use afl-fuzz as a traditional, "dumb" fuzzer.
    For that, you can use the -n option - but expect much worse results.)

[-] PROGRAM ABORT : No instrumentation detected
         Location : check_binary(), afl-fuzz.c:6959
```

FIGURE B.1: Example of an AFL error message when no instrumentation is detected

If a participant decided not to write a fuzz target but to fuzz one of the executables of the target programs, we call that fuzz-target-less fuzzing (FTL fuzzing). In the case that FTL fuzzing was used in AFL, it spans from Selected Function to Built FuzzTarget. In this case the target executable is noted in the columns.

B.5 User Rating of Fuzzer

Table B.3 shows the Usability and Overall rating given by our participants for AFL and libFuzzer.

Overall										Usability									
Fuzzer	1	2	3	4	5	6	7	plot	median	Fuzzer	1	2	3	4	5	6	7	plot	median
AFL	0	1	0	5	3	5	3		5	AFL	1	3	1	3	2	4	3		5
libFuzzer	0	1	4	4	5	3	0		4	libFuzzer	0	2	3	3	3	6	0		5

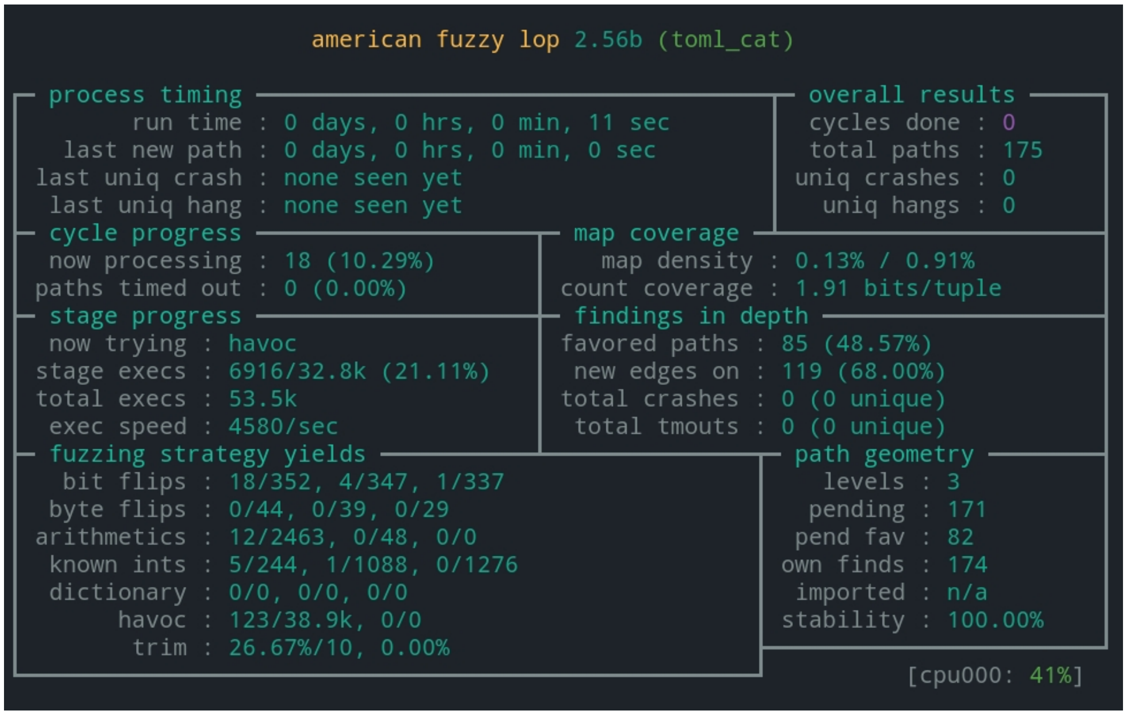
TABLE B.3: Usability and Overall rating of AFL and libFuzzer given by the participants

B.6 Example of an Error Message

In Figure B.1 an error message of AFL is shown when no instrumentation in the target is detected.

B.7 Outputs of Fuzzers While Running

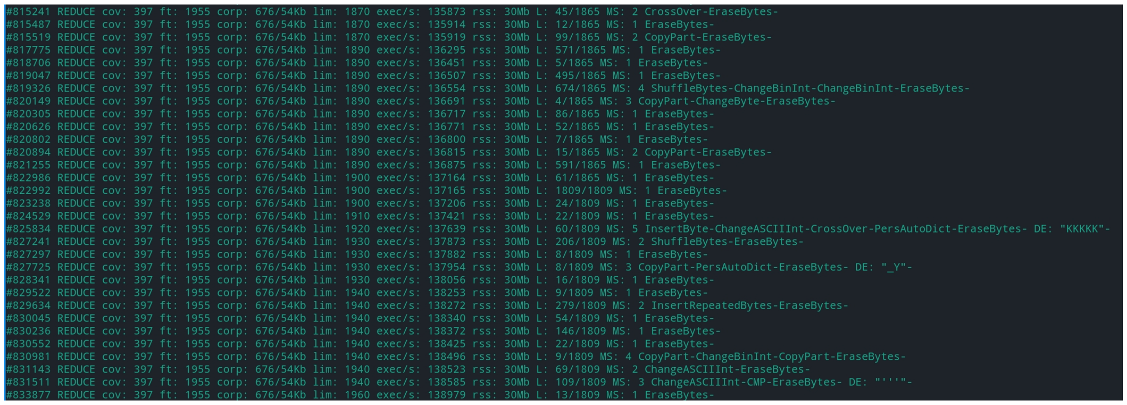
Figures B.2 and B.3 show an example of the outputs while running of AFL and libFuzzer.



```
                   american fuzzy lop 2.56b (toml_cat)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 11 sec       │  cycles done : 0     │
│   last new path : 0 days, 0 hrs, 0 min, 0 sec        │  total paths : 175   │
│ last uniq crash : none seen yet                      │ uniq crashes : 0     │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 18 (10.29%)       │    map density : 0.13% / 0.91%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.91 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : havoc                 │ favored paths : 85 (48.57%)           │
│ stage execs : 6916/32.8k (21.11%)   │  new edges on : 119 (68.00%)          │
│ total execs : 53.5k                 │ total crashes : 0 (0 unique)          │
│  exec speed : 4580/sec              │  total tmouts : 0 (0 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 18/352, 4/347, 1/337                  │    levels : 3          │
│  byte flips : 0/44, 0/39, 0/29                      │   pending : 171        │
│ arithmetics : 12/2463, 0/48, 0/0                    │  pend fav : 82         │
│  known ints : 5/244, 1/1088, 0/1276                 │ own finds : 174        │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 123/38.9k, 0/0                        │ stability : 100.00%    │
│        trim : 26.67%/10, 0.00%                      ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000: 41%]
```

FIGURE B.2: Example of an AFL output while running



```
#815241 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1870 exec/s: 135873 rss: 30Mb L: 45/1865 MS: 2 CrossOver-EraseBytes-
#815487 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1870 exec/s: 135914 rss: 30Mb L: 12/1865 MS: 1 EraseBytes-
#815519 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1870 exec/s: 135919 rss: 30Mb L: 99/1865 MS: 2 CopyPart-EraseBytes-
#817775 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1890 exec/s: 136295 rss: 30Mb L: 571/1865 MS: 1 EraseBytes-
#818706 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1890 exec/s: 136451 rss: 30Mb L: 5/1865 MS: 1 EraseBytes-
#819047 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1890 exec/s: 136507 rss: 30Mb L: 495/1865 MS: 1 EraseBytes-
#819326 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1890 exec/s: 136554 rss: 30Mb L: 674/1865 MS: 4 ShuffleBytes-ChangeBinInt-ChangeBinInt-EraseBytes-
#820149 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1890 exec/s: 136691 rss: 30Mb L: 4/1865 MS: 3 CopyPart-ChangeByte-EraseBytes-
#820305 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1890 exec/s: 136717 rss: 30Mb L: 86/1865 MS: 1 EraseBytes-
#820626 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1890 exec/s: 136771 rss: 30Mb L: 52/1865 MS: 1 EraseBytes-
#820802 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1890 exec/s: 136800 rss: 30Mb L: 7/1865 MS: 1 EraseBytes-
#820894 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1890 exec/s: 136815 rss: 30Mb L: 15/1865 MS: 2 CopyPart-EraseBytes-
#821255 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1890 exec/s: 136875 rss: 30Mb L: 591/1865 MS: 1 EraseBytes-
#822986 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1900 exec/s: 137164 rss: 30Mb L: 61/1865 MS: 1 EraseBytes-
#822992 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1900 exec/s: 137165 rss: 30Mb L: 1809/1809 MS: 1 EraseBytes-
#823238 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1900 exec/s: 137206 rss: 30Mb L: 24/1809 MS: 1 EraseBytes-
#824529 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1910 exec/s: 137421 rss: 30Mb L: 22/1809 MS: 1 EraseBytes-
#825834 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1920 exec/s: 137639 rss: 30Mb L: 60/1809 MS: 5 InsertByte-ChangeASCIIInt-CrossOver-PersAutoDict-EraseBytes- DE: "KKKKK"-
#827241 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1930 exec/s: 137873 rss: 30Mb L: 206/1809 MS: 2 ShuffleBytes-EraseBytes-
#827297 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1930 exec/s: 137882 rss: 30Mb L: 8/1809 MS: 1 EraseBytes-
#827725 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1930 exec/s: 137954 rss: 30Mb L: 8/1809 MS: 3 CopyPart-PersAutoDict-EraseBytes- DE: "_Y"-
#828341 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1930 exec/s: 138056 rss: 30Mb L: 16/1809 MS: 1 EraseBytes-
#829522 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1940 exec/s: 138253 rss: 30Mb L: 9/1809 MS: 1 EraseBytes-
#829634 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1940 exec/s: 138272 rss: 30Mb L: 279/1809 MS: 2 InsertRepeatedBytes-EraseBytes-
#830045 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1940 exec/s: 138340 rss: 30Mb L: 54/1809 MS: 1 EraseBytes-
#830236 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1940 exec/s: 138372 rss: 30Mb L: 146/1809 MS: 1 EraseBytes-
#830552 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1940 exec/s: 138425 rss: 30Mb L: 22/1809 MS: 1 EraseBytes-
#830981 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1940 exec/s: 138496 rss: 30Mb L: 9/1809 MS: 4 CopyPart-ChangeBinInt-CopyPart-EraseBytes-
#831143 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1940 exec/s: 138523 rss: 30Mb L: 69/1809 MS: 2 ChangeASCIIInt-EraseBytes-
#831511 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1940 exec/s: 138585 rss: 30Mb L: 109/1809 MS: 3 ChangeASCIIInt-CMP-EraseBytes- DE: "''''"-
#833877 REDUCE cov: 397 ft: 1955 corp: 676/54Kb lim: 1960 exec/s: 138979 rss: 30Mb L: 13/1809 MS: 1 EraseBytes-
```

FIGURE B.3: Example of a libFuzzer output while running

B.8 Improvements to the Fuzzing Process

Most participants tried a variety of improvements which are described in the following. The improvements used by the participants are depicted in Table B.4 for the first task and Table B.5 for the second task. The column Computing Power describes what participants did to give more computing power to the fuzzing process or if they tried to parallel the process. If participants made significant changes to the fuzz target, it is depicted in the column Fuzz Target. Moreover, the columns Corpus and Dict show if participants used meaningful seeds in the corpus or a dictionary, respectively. The sanitizers used by the participants are depicted in the column Sanitizer. In the last column, Other, other notable techniques used are shown. If an element is followed by an *, it indicates that it is based on support.

Moreover, the Tables B.6 and B.7 show the ratio and percentage of participants that used the mutual and distinct fuzzer improvements.

In the following, we will give further insides into the improvements of the fuzzing process made by the participants.

Participant	Fuzzer	Program	Computing Power	Fuzz Target	Corpus	Dict	Sanitizer	Other
P26	AFL	tomlc99						AFL_HARDEN
P06	AFL	tomlc99			x			AFL_HARDEN
P14	AFL	tomlc99	2 instances		x		ASAN*	
P02	AFL	tomlc99	3 instances		x	x	ASAN, MSAN	
P33	AFL	libroxml			x			afl-cmin
P05	AFL	libroxml			x		ASAN (failed)	
P21	AFL	libroxml			x		ASAN	
P25	AFL	libroxml			x			
P41	AFL	libroxml			x			
P45	AFL	libroxml			x			
P08	libFuzzer	tomlc99	20 instances		x		ASAN	
P44	libFuzzer	tomlc99						debug_symbols
P12	libFuzzer	tomlc99	20 instances		x		ASAN*, signed-integer-overflow*	debug symbols, -ignore_crashes=1 -fork=4
P48	libFuzzer	tomlc99	8 instances				ASAN, signed-integer-overflow, USBAN	value_profile=1, coverage visualization (failed)
P16	libFuzzer	tomlc99		error correction	x	x	ASAN*, signed-integer-overflow*	coverage visualization (failed)
P19	libFuzzer	libroxml	4 instances	error correction*	x			-only_ascii=1, -detect_leaks=0
P15	libFuzzer	libroxml	10 instances	error correction	x	x	ASAN	

TABLE B.4: Improvements Task 1

Participant	Fuzzer	Program	Computing Power	Fuzz Target	Corpus	Dict	Sanitizer	Other
P19	AFL	tomlc99	2 instances		x		ASAN*, MSAN*	afl-tmin
P15	AFL	tomlc99	4 instances		x	x		afl-cmin, afl-tmin, debug symbols
P48	AFL	libroxml			x		ASAN	
P08	AFL	libroxml			x			
P44	AFL	libroxml			x			
P12	AFL	libroxml			x		ASAN	
P16	AFL	libroxml			x	x	ASAN	AFL_HARDEN
P25	libFuzzer	tomlc99			x		ASAN*	debug symbols, -fork=1 -ignore_crashes=1
P21	libFuzzer	tomlc99	4 instances		x		ASAN	-detect_leaks=0
P33	libFuzzer	tomlc99		error correction	x			
P41	libFuzzer	tomlc99	4 instances		x		ASAN, signed-integer-overflow	debug symbols, -fsanitize-recover=address ASAN_OPTIONS=halt_on_error=0
P45	libFuzzer	tomlc99	1000 instances		x			debug symbols
P05	libFuzzer	tomlc99	50 instances	error correction	x		ASAN	detect_leaks=0
P26	libFuzzer	libroxml						
P06	libFuzzer	libroxml			x		ASAN*, signed-integer-overflow*	
P02	libFuzzer	libroxml					ASAN, signed-integer-overflow, MSAN	-detect_leaks=0, debug symbols
P14	libFuzzer	libroxml	4 instances	error correction	x		ASAN	Bash-script to skip crashes, ignore_crashes (failed), detect_leaks=0, debug symbols

TABLE B.5: Improvements Task 2

Computing Power A widespread improvement for both fuzzer was to improve the process by enhancing the computing power. The most common ways to do this were running the fuzzer for a more extended period and using the appropriate form of parallelization. The extended period was usually between one and three hours during the day or several hours overnight. For AFL and libFuzzer, parallelization was not an issue for the participants. Some participants upgraded the virtual machine or used other external machines to get more computing power. Neither of the participants reported a clear improvement from computing power. We suspect that more computing power was chosen as an improvement as it was recommended in both documentations and was fairly easy to do.

Improvements to the Fuzz Target Many Participants tried to use more fuzz targets and, in that sense, tried to fuzz more target functions to find more bugs. The targeting of more functions was independent of the fuzzer used. They also tried to improve their existing targets, but this was mostly done when using libFuzzer. The improvement to existing targets was nearly solely based on the situation that the participants had problems giving a useful input to the target function, which resulted in false positives and, therefore, the necessity to remove the problem. The root of the problems was that the input to the target programs had to be a null-terminated string, but the input given from libFuzzer is not null-terminated. The need to fix the fuzz target was not a problem when using AFL because the input for the target program is null-terminated.

Compiler Optimization for AFL As depicted in Table B.7, only 35% of the 17 participants used the afl-clang-fast compiler, which results in a speed increase of between 10% and 200% in comparison to the vanilla AFL compiler. Half of those participants actively changed the compiler to afl-clang-fast from one of the two slower compilers, afl-gcc and afl-clang. In comparison, three participants directly started with the fastest possible compiler. The changes necessary to use the fastest compiler are minimal. Instead of setting the compiler to afl-gcc or afl-clang, the compiler must be set to afl-clang-fast. The fastest compiler can be used on nearly every target program. Only on rare occasions does the compilation process with afl-clang-fast fail and a slower compiler has to be chosen. As a consequence, setting the compiler to afl-clang-fast is highly recommended. The insufficient documentation could explain that only a handful of participants used afl-clang-fast since the necessary changes from a technical and usability perspective are minimal.

Error Suppression in libFuzzer Some participants stated that they improved their fuzzing by suppressing error messages from libFuzzer. They can be divided into two groups. The first group, which contained two participants, was annoyed by the leak detection. All leaks are self-induced due to misconfiguration in the fuzz target. We were not able to find any leaks in the target program. One of those two participants, participant P05, thought about the situation and turned the leak detection right back on because they thought leaks could result in bugs. They stated:

> So at the very beginning I had the leaks disabled, or when running, set the leaks flag to zero. But then I quickly took it back because

memory consumption or if memory leaks can lead to a DOS sooner or later

The other participant was not really aware of the situation and tried different combinations of sanitizers and leak detection which resulted in different findings for them. They were not able to make any sense out of the situation and just started to use a debugger to get a better understanding.

The other group of three participants deactivated the address sanitizer because the usage of the address sanitizer resulted in a finding of the fuzzer. Although they acknowledged the finding, they wanted to get more findings, but the crash that occurred while using the address sanitizer was so dominant that no other crashes were found. They concluded that the address sanitizer was the cause of the situation and therefore dropped the address sanitizer to get more findings. Participant P41 also believed that while the crash produced with the address sanitizer is valid, the crash can not be triggered in a realistic situation because it was not produced without the address sanitizer and consequently ignored it.

Fuzzers point of view When looking at the accumulated numbers of mutual improvements for the respective fuzzer, shown in Table B.6, it can be seen that the usage of a meaningful corpus is the most common improvement for both fuzzers. In addition, improvements for the fuzzing process with AFL are rather sparse. For libFuzzer, the utilization of more than one core and the usage of the address sanitizer are also expected improvements.

Improvement	AFL		libFuzzer	
Corpus	16/17	94%	13/13	100%
Dict	3/3	100%	2/2	100%
Multi Proc	4/4	100%	10/10	100%
ASAN	5/6	83%	9/9	100%
MSAN	1/1	100%	1/1	100%

TABLE B.6: Ratio and success percentage of mutual fuzzer improvements

Assessing the fuzzer-specific improvements, as shown in Table B.7, the percentage of used improvements is even lower with no defining improvements.

Improvement	AFL		libFuzzer	
Compiler opti.	4/4	100%	-	
Best compiler	2/2	100%	-	
Pers. Mode	0/0	-	-	
AFL_HARDEN	3/3	100%	-	
afl-tmin	2/2	100%	-	
afl-cmin	2/2	100%	-	
Debug Symbols	-		7/7	100%
Coverage Vis.	-		0/2	0%
UBSAN	-		1/1	100%
Signed San	-		3/3	100%
ignore crashes	-		2/3	67%
value profile	-		1/1	100%

TABLE B.7: Ratio and success percentage of distinct fuzzer improvements

Appendix C

Replication: AFL vs. libFuzzer

C.1 Pre-Questionnaire

Programming Experience

- How many years of programming experience do you have?
  [Number]
- Which programming languages are you proficient in?
  Java, C, C++, Python, Ruby, Go, Pascal, Basic, Rust, Perl, PHP, JavaScript, R, Other (please specify) [Free text]
- Please rate how proficient you are in the following programming languages.
  1 - not at all - 7 - excellent
  Java, C, C++, Python, Ruby, Go, Pascal, Basic, Rust, Perl, PHP, JavaScript, R

Static Code Analysis

- Have you ever used static code analysis to support your programming?
  Yes, No
- *if yes*
  - Which static code analysis tools have you ever used?
    [Free text]
  - Please rate your agreement with the following statement: static code analysis supported my programming!
    1 - Disagree very strongly - 7 - agree very strongly

Dynamic Code Analysis

- Are your familiar with the term ’fuzzing’?
  Yes, No

- Have you ever used a fuzzer?
  Yes, No
- *if yes*
  - Please name all fuzzers you have ever used.
    [Free text]
  - Have you ever found a bug with a fuzzer?
    Yes, No

Demographics

- Please select your gender.
  Male, Female, Prefer not say, Prefer to self-describe: [Free text
- Please enter your age:
  [Number]
- Which country do you live in?
  [Free text]
- Do you have a university degree?
  Yes, No

Degree

- *if yes*
  - From which university/universities is/are your degree(s) from?
    [Free text]
  - What degree(s) do you have?
    [Free text]

Occupation

- What is your current occupation? (Multiple answers are allowed)
  Industry developer, Freelance tester, Industry tester, Industry researcher, Academic researcher, Undergraduate student, Graduate student, Other: [Free text]

Education

- *if Undergraduate student or Graduate student*
  - At which university are you enrolled?
    [Free text]
  - What is your subject?
    [Free text]

Development

- What type(s) of software do you develop/test? (Multiple answers allowed)
  Web applications, Mobile/App applications, Desktop applications, Embedded Software Engineering, Enterprise applications, Other: [Free text]
- How many years of experience do you have with software development/testing in general?
  [Number]

Skills

- How did you gain your IT skills? (Multiple answers allowed)
  At university, On the job, Free online courses, Paid online courses, Paid on-location courses, Free on-location courses, Self-taught, Other [Free text]
- How did you gain your IT-security skills? (Multiple answers allowed)
  At university, On the job, Free online courses, Paid online courses, Paid on-location courses, Free on-location courses, Self-taught, Other [Free text]
- What is currently your main source of learning about IT-security?
  Mandatory IT-security trainings, Voluntary IT-security trainings, Colleagues, Websites: [Free text], Journals / magazines: [Free text], Other: [Free text]

Company

- How many people work in your team? Please enter 1 if you work on your own.
  [Number]
- Please select what is more important to you. *0 - Functionality - 100 - Security*

C.2 Post-Questionnaire

The following question were asked for AFL as well as libFuzzer if not otherwise stated. The questions were present to the participant in order of working with the fuzzers.

Fuzzing Steps

Fuzzer Task - Start

- How did you complete the following subtasks when using *fuzzer*?
  I did not work on it, successfully, unsuccessfully, I don't know
  Subtask 1 - Find a suitable function to fuzz.; Subtask 2 - Write your fuzz

target in an external file.; Subtask 3 - Compile and instrument the program.; Subtask 4 - Compile the fuzz target.; Subtask 5 - Run the fuzzer and interpret the output.; Subtask 6 - If necessary, adjust and improve.

- What was the last thing you worked on when using *fuzzer*?
  [Free text]

***Fuzzer* Task - Overview Target Program**

- What was the target program when you used *fuzzer*?
  [Free text]
- How well were you able to get an overview of the target program on a scale from 1 to 7 (1 very bad, 7 very good)?
  1 - very bad - 7 - very good
- *if fuzzer was AFL*
  - How did you use AFL?
    I wrote a fuzz target, I used AFL on the executable without a fuzz target, Both, I don't know, I did not use AFL

***Fuzzer* Task - Function Selection**

- Which function(s) did you chose to fuzz?
  [Free text]
- What was the reason that you chose this/these function(s)?
  [Free text]

***Fuzzer* Task - Building and Instrumentation - Target Program**

- Which compiler(s) did you use to compile the target program when using *fuzzer*?
  AFL-gcc, AFL-clang, AFL-clang-fast, clang, gcc, CC, g++, clang++, I don't know, Other: [Free text]
- Please copy and paste the flags or environmental variables you used for compiling the target program when using AFL.
  [Free text]
- Did you use instrumentation on the target program when using AFL?
  Yes, No, I don't know

***Fuzzer* Task - Building and Instrumentation - Fuzz Target**

- Which compiler(s) did you use to compile the fuzz target when using AFL?
  AFL-gcc, AFL-clang, AFL-clang-fast, clang, gcc, CC, g++, clang++, I don't know, Other: [Free text]
- Please copy and paste the flags or environmental variables you used for compiling the fuzz target when using *fuzzer*.
  [Free text]
- Did you use instrumentation on the fuzz target when using *fuzzer*?
  Yes, No, I don't know

***Fuzzer* Task - Running the Fuzzer**

- Did you start *fuzzer*? *Yes, No, I don't know*
- Did you see an output similar to this *picture of output* when using *fuzzer*?
  Yes, No, I don't know
- How would you rate the Output of *fuzzer* while running on a scale from 1 to 7 (1 very bad, 7 very good)?
  1 - very bad - 7 - very good
- How did you judge whether *fuzzer* was running well?
  [Free text]

***Fuzzer* Task - Improvements**

- What improvements to your fuzzing did you make?
- if AFL
  - *Corpus, Dictionary, Multiple Fuzzing Instances, Address Sanitizer, Memory Sanitizer, Used afl-clang-fast, Used Persistent Mode, AFL_HARDEN, afl-tmin, afl-cmin, None, Other: [Free text]*
- if libFuzzer
  - *Corpus, Dictionary, Multiple Fuzzing Instances, Address Sanitizer, Memory Sanitizer, Debug Symbols, Coverage Visualization, Undefined Behavior Sanitizer, Signed Integer Overflow Sanitizer, ignore_crashes, value profile, leaks off, None, Other: [Free text]*
- Please rate the benefit of your improvements for your fuzzing from the highest, top, to lowest, bottom, when using AFL.

- *Sorting of improvements chosen above*

***Fuzzer* Task - Crashes**

- Did *fuzzer* report a crash to you?
  Yes, No
- How would you rate the output of *fuzzer* for a crash on a scale from 1 to 7 (1 very bad, 7 very good)?
  1 - very bad - 7 - very good
- How many crashes did *fuzzer* report to you? *[Number]*
- How many bugs in the target program did you find?
  [Number]
- How did you confirm that a crash was a bug?
  Manually inspecting the code, Executing the target program with the crashing input, Debugger, None, Other: [Free text]

***Fuzzer* Task - Summary**

- Please rate the difficulty of the subtasks from the highest, top, to lowest, bottom, when using *fuzzer*.
  Sorting of Subtasks

***Fuzzer* Task - Scores**

- How would you rate *fuzzer* overall on a scale from 1 to 7 (1 very bad, 7 very good)?
  1 - very bad - 7 - very good
- How would you rate the usability of *fuzzer* on a scale from 1 to 7 (1 very bad, 7 very good)?
  1 - very bad - 7 - very good
- How would you rate the documentation of AFL on a scale from 1 to 7 (1 very bad, 7 very good)?
  1 - very bad - 7 - very good
- What are your final thoughts about the usability of *fuzzer*? *[Free text]*

Fuzzing Knowledge

- Please explain what "Instrumentation" means in the context of fuzzing?
  [Free text]

C.2.1 Comparison

- Which fuzzer would you prefer to use for the following actions?
  0 - prefer AFL - 100 - prefer libFuzzer
  Writing a fuzz target, Building and instrumenting, Running the fuzzer, Monitoring output while running, Interpreting output on crash, Improving the fuzzing process
- Please rate which fuzzer you think is better in the following aspects
  0 - AFL better - 100 - libFuzzer better
  Helpfulness of error messages, Documentation, Supporting the user, Correctness of output, Transparency of actions, Working correctly
- Please rate which fuzzer made you feel the following emotions more
  0 - AFL - 100 - libFuzzer
  Anger, Enjoyment, Frustration, Surprise
- Which of the fuzzers would you use in the future?
  AFL, libFuzzer, Both, None

C.3 Nationalities

Nation	**n**	**Nation**	**n**
Argentina	1	Bangladesh	2
Brazil	2	Cameroon	1
China	1	Colombia	1
Czech Republic	1	Ecuador	2
Egypt	4	France	1
Greece	2	Hungary	1
India	10	Iran	1
Italy	2	Madagascar	1
Malaysia	1	Mexico	1
Morocco	1	Pakistan	5
Peru	1	Philippines	1
Russia	1	Slovakia	1
Spain	2	Spanish	1
Sri Lanka	1	Thailand	1
Tunisia	2	Turkey	3
United States	3	Venezuela	2
Vietnam	1		

TABLE C.1: Current country of our participants.

Appendix D

Own vs. Other

D.1 How to solve the task

Since only one function had to be implemented by the participants and by the design of the task, the function was also easy to fuzz. It is a suitable choice for fuzzing the extension.

A fuzz target could look like shown in Listing D.1.

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdio.h>

#include "readInRC.h"
#include "shell.h"



int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    char* input = malloc(Size+1);
    memcpy(input, Data, Size);
    input[Size] = '\0';

    scanRC(input);

    free(input);

    return 0;
}
```

LISTING D.1: Fuzz Target for scanRC

The Data argument provided by the fuzzer is taken and the content is written to an afor created memory block. Most importantly, the memory chunk is null-terminated to create a string. This string is then inserted into the participant's written function. In the end, the allocated memory is freed.

One way to build all necessary code is to build the shell with instrumentation firstly and, in a second step, build and link the fuzz target. The shell and the fuzz target can be build and instrumented with the commands shown in Listing D.2.

```
# Building Shell
make CC=clang CFLAGS="-fsanitize=fuzzer-no-link,address"


# Building Fuzz Target
clang target.c -fsanitize=fuzzer,address -I./ ... shell.a -o target


#Running the fuzzer
./target
```

LISTING D.2: Build commands of libFuzzer for libroxml

For instrumenting the shell, the compiler clang has to be used and the fsanitize-flag has to be set to *fuzzer-no-link*. To build the fuzz target also clang has to be used, and, again, the fsanitize-flag has to be set, but this time including the linking to the fuzzer. Therefore the flag is set to *fuzzer*. In this example, the *address* flag is also set to activate the address sanitizer, which helps to identify memory-related bugs and also provides additional information in the case that a crash occurs.

Starting the fuzzer is as simple as running the resulting executable of the building and instrumentation process. It is also shown in Listing D.2.

After the fuzzer is started, it periodically gives a status report of the fuzzing run. When a crash is found, the fuzzer stops and reports information associated with the crash. Depending on the type of crash and the flags used, such as debug symbols and the address sanitizer, different information is given. Among other things, the following can also be specified, the type of crash, the location of the crash and the input that led to the crash. This information can then be used to locate and fix the bug.

Improvements to the fuzzing process can be situational. Nevertheless, the usage of different sanitizers, a corpus or dictionary and having a solid understanding of the code covered are nearly always reasonable.

D.2 Dependencies in Fuzzing Process

For the dependencies, we divided the building of all necessary code into three parts. The first part is the creation of a fuzz target that is *buildable* in the sense that no error is thrown when the right compilation and linking commands are executed. The second part is the creation of a *meaningful* fuzz target. This includes actually fuzzing a function of the target program or at least showing a strong intention in doing so. The third part is writing a *triggerable* fuzz target. A triggerable fuzz target calls a meaningful function in a way that interesting parts of the target function are reached and potential startup initialization are handled. We defined the following dependencies, which can also be seen in Figure D.1.

The steps of selecting a suitable function to fuzz, writing a *buildable* fuzz target, the instrumentation and using ASAN are not dependent on any of the other steps.

Creating a corpus or writing a dictionary is dependent on the selected function. It is not preferable to have a corpus consisting of HTTP requests when fuzzing an image parser.

To be able to build all necessary code, the fuzz target needs to be buildable. Moreover, we decided that a dependency on a *meaningful* fuzz target for building all necessary code was appropriate because the building process can, for example, be much easier if no function of the target program is fuzzed. Building all necessary code is the dependency for running the fuzzer.

To have a *triggerable* fuzz target it needs to be *meaningful*.

For successfully finding and recognizing a bug, we defined a *triggerable* fuzz target and running the fuzzer as the dependencies. Moreover, the usage of a fitting corpus or dictionary and the address sanitizer, as well as instrumenting all necessary code, can be necessary to be able to be successful and are thus dependent dependencies. They are marked with dashed lines.

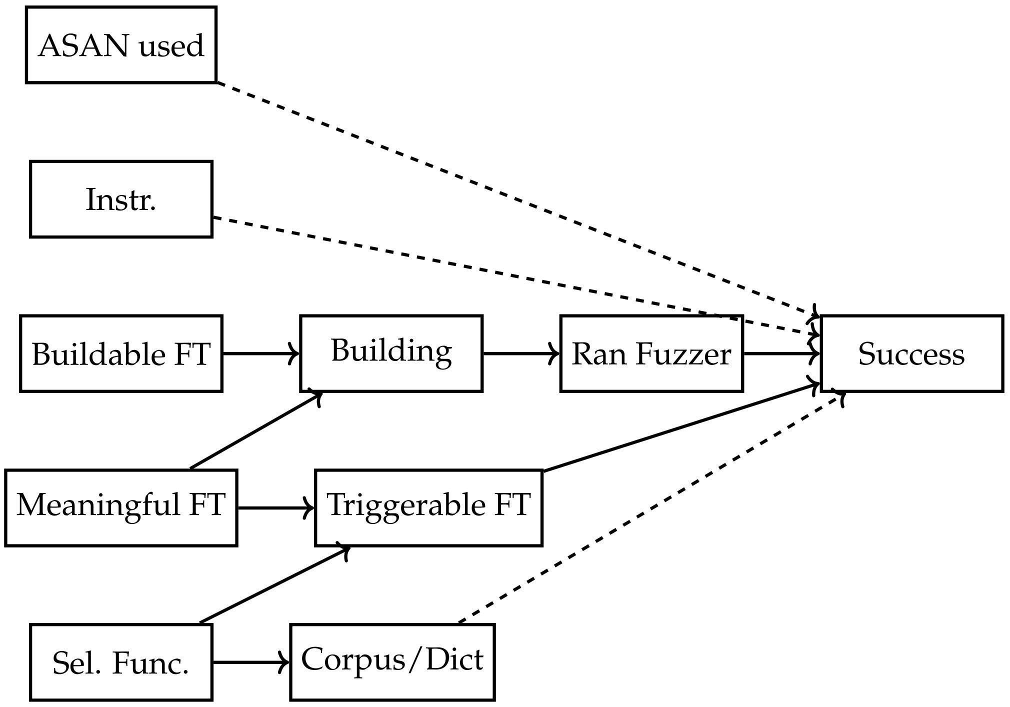


FIGURE D.1: Dependencies of steps