A Qualitative Usability Evaluation of the Clang Static Analyzer and libFuzzer with CS Students and CTF Players

Stephan Plöger, *Fraunhofer FKIE;* Mischa Meier, *University of Bonn;*
Matthew Smith, *University of Bonn, Fraunhofer FKIE*

https://www.usenix.org/conference/soups2021/presentation/ploger

This paper is included in the Proceedings of the Seventeenth Symposium on Usable Privacy and Security.

August 9–10, 2021

978-1-939133-25-0

A Qualitative Usability Evaluation of the Clang Static Analyzer and libFuzzer with CS Students and CTF Players

Stephan Plöger
Fraunhofer FKIE

Mischa Meier
University of Bonn

Matthew Smith
University of Bonn, Fraunhofer FKIE

Abstract

Testing software for bugs and vulnerabilities is an essential aspect of secure software development. Two paradigms are particularly prevalent in this domain: static and dynamic software testing. Static analysis has seen widespread adoption across the industry, while dynamic analysis, in particular fuzzing, has recently received much attention in academic circles as well as being used very successfully by large corporations such as Google, where for instance, over 20,000 bugs have been found and fixed in the Chrome project alone. Despite these kinds of success stories, fuzzing has not yet seen the kind of industry adoption static analysis has.

To get first insights, we examine the usability of the static analyzer Clang Static Analyzer and the fuzzer libFuzzer. To this end, we conducted the first qualitative usability evaluation of the Clang Static Analyzer [6] and libFuzzer [16]. We conducted a mixed factorial design study with 32 CS masters students and six competitive Capture the Flag (CTF) players. Our results show that our participants encountered severe usability issues trying to get libFuzzer to run at all.

In contrast to that, most of our participants were able to run the Clang Static Analyzer without significant problems. This shows that, at least in this case, the usability of libFuzzer was worse than of the Clang Static Analyzer. We make suggestions on how libFuzzer could be improved and how both tools compare.

USENIX Symposium on Usable Privacy and Security (SOUPS) 2021.
August 8–10, 2021, Virtual Conference.

1 Introduction

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 [13]. 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: [19,29,31–35,38,41–44,46,47,49–51,53,54,58,60–65,67,69, 78–87]. Moreover, large software companies such as Google, Microsoft, Cisco and other use fuzzing very successfully, for instance, using fuzzing Google found over 20,000 bugs in Chrome alone [3]. 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. [28] and libFuzzer because it is a popular example of a dynamic code analysis tool in academia [46, 55, 66]. 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 of 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.

Supplementary to the Appendix, we provide additional information in a companion document which can be found here: https://uni-bonn.sciebo.de/s/dUH7FOedjHbG5vy.

2 Related Work

The related work section is divided into two parts: usability evaluations of static analysis tools and study methodology concerning developer studies. To the best of our knowledge there are no studies concerning the usability of fuzzers or a usability comparison of static analysis and fuzzing.

Static Analysis Studies Smith et al. [72] conducted a heuristic walkthrough and a user study about 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 of 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 [73, 74]. They wanted to study the 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 but not to run the tool itself. Based on their finding they gave recommendations for the design of static analysis tools. Their main suggestion was that tools should help developers search for relevant web resources. Our study goes beyond this work, since they actually had to use the tool and had more time to do so.

In 2013 Johnson and Song conducted 20 interviews about static code analysis with 20 developers [48]. 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. In 2016 Christakis and Bird conducted a survey at Microsoft to get more insights into the use of static code analysis [37]. They set the focus on the barriers of 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 and desired customizability and the option to prioritize warnings. Vassallo et al. confirmed those findings in 2018 [25].

Sadowski and colleges presented a set of guiding principles for their program analysis tool Tricorder, a program analysis platform developed for Google. They included an empirical in-situ evaluation emphasizing that developers do not like false positives and that workflow integration is key [68].

A comparison of open-source static analysis tools for C/C++ code was done by Arusoaie et al. in 2017 [28]. They compared 11 analysis tools on the Toyota ITC test suite [70]. They ranked them by productivity which balances the detection rate with the false-positive rate to compensate for a high false-positive rate. The top three performers were clang, Frama-C [12] and OCLint [20].

Study Methodology Since it is difficult to recruit professional developers [26, 27, 71], Naiakshina conducted a study to evaluate CS students' use in developer studies [57]. They found that for their password storage study students were a viable proxy for freelance developers. Naiakshina followed this up with a comparison to professional developers in German companies [56]. Here they found that the professional developers preformed better overall than students, but that the effects of the independent variables on the dependent ones held none the less and thus conclude that CS students could be used for comparative studies in their case.

A study by Votipka showed that taking part in CTF games tends to have a positive effect on security thinking [76] and hackers are comparable to testers in software vulnerability discovery processes [77].

3 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.

3.1 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 [17], Coverity [9], CodeSonar [7] and checkmarx [4]. Unfortunately, they all forbid publishing evaluations in their terms of use [5, 8, 10, 18]. We based our selection of the open-source static analysis tool on the evaluation of Arusoaie et al. [28]. 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 [1], AFL++ [2], libFuzzer, honggfuzz [14] and radamsa [21]. Our literature review showed that AFL/AFL++ and its forks, as well as libFuzzer, are the most common fuzzers in use [30, 31, 36, 40, 46, 52, 55, 78]. Both AFL and libFuzzer were viable choices. While both Fuzzers can fulfil 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 fuzzes code by using the executable of the target program. 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.

3.2 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 datasets like the DARPA Cyber Grand Challenge [11] were not viable options for us. We also decided against using tools like Lava-M [39] 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 5. 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 [15] for the easy task and two bugs into a local copy of the open-source project Tesseract [23] 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. A detailed description of the bugs can be found in the companion document.

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 [24] 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 [22] 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 is 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, it was not possible to trigger the bugs with the AFL fuzz targets. Details on the bugs can be found in the companion document.

3.3 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 the 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 degree in computer science, had some basic knowledge on 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 13 of the Appendix shows the demographics of the CS student participants.

Only six out of the 32 participants reported that they have 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.

3.4 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.

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 14 in the Appendix. 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.

3.5 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.

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 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 learnt to overcome them.

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 group, 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 if 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.

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

5 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.

6 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 Appendix A.1 and in the companion document. 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 [75] with two researchers. The two researchers started with 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 recoded 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. The final coding set can be found in the companion document.

6.1 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 1 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 artefact 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 11 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.

6.2 Static Task

The results of the static analysis tasks can be seen in Table 1. Table 6 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. Readers unfamiliar with this topic can find additional information in the companion document.

Step1: Build Target Program with Clang Static Analyzer 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 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.

6.3 Dynamic Task

The results of the dynamic analysis tasks in Table 1 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 9 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 was roughly half, just like for the hard static task.

Unlike with the Clang Static Analyzer, which almost all participants used correctly, we found many problems with the usage of libFuzzer. Table 2 gives an overview of where participants had problems. The columns of Table 2 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 grey 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.

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. Readers unfamiliar with these steps can find additional information in the companion document.

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 losing interest.

Step 1: Find a Suitable Function 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: [...].

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 1: CS static analysis and fuzzing overview

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

> 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 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 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. None of them was able to find a solution for this, as depicted in the companion document. 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 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.

7 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 A.2 and in the companion document.

An overview of the CTF-group's success can be seen in Table 3. 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.

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 artefact 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*.

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

Table 3: CTF: overall results

Dynamic Analysis

Despite being more experienced and security savvy, our CTF participants also had trouble with libFuzzer. Table 4 gives an overview of where participants had problems.

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 a 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 a 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 of the 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

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

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 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 to implement 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.

8 Discussion

8.1 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 [59], 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 [45] 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.

8.2 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.

8.3 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 to 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 to find 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 11 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 tolerating the poor usability of libFuzzer due to its better perceived utility.

9 Conclusion and Future Work

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. 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.

Acknowledgments

The authors would like to thank Sirko Höer for helping with the fuzzing task selection. This work was partially funded by the ERC Grant 678341: Frontiers of Usable Security.

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

References

[1] Afl. https://github.com/google/AFL. Accessed: 02-13-21.

[2] Afl++. https://github.com/AFLplusplus/AFLplusplus. Accessed: 02-13-21.

[3] Bugs found in chrome with fuzzing. https://bugs.chromium.org/p/chromium/issues/list?can=1&q=label%3AClusterFuzz+-status%3AWontFix%2CDuplicate&colspec=ID+Pri+M+Stars+ReleaseBlock+Component+Status+Owner+Summary+OS+Modified&x=m&y=releaseblock&cells=ids. Accessed: 02-13-21.

[4] Checkmarx sast. https://www.checkmarx.com/de/products/static-application-security-testing. Accessed: 02-13-21.

[5] Checkmarx sast license agreement. https://checkmarx.atlassian.net/wiki/spaces/CCD/pages/1253442222/CxIAST+End+User+License+Agreement+EULA. Accessed: 02-13-21.

[6] Clang static analyzer. https://clang-analyzer.llvm.org/. Accessed: 02-13-21.

[7] Codesonar sast. https://www.grammatech.com/codesonar-cc. Accessed: 02-13-21.

[8] Codesonar sast license agreement. https://support.grammatech.com/documentation/licenses/GrammaTech_License_Agreement_CodeSonar_ver.2016.1.0.pdf. Accessed: 02-13-21.

[9] Coverity scan. https://scan.coverity.com/. Accessed: 02-13-21.

[10] Coverity scan license agreement. https://www.synopsys.com/company/legal/software-integrity/coverity-product-license-agreement.html. Accessed: 02-13-21.

[11] Darpa cyber grand challenge. https://www.darpa.mil/program/cyber-grand-challenge. Accessed: 02-13-21.

[12] Frama-c. https://frama-c.com/. Accessed: 02-13-21.

[13] Gartner magic quadrant for application security testing. https://www.gartner.com/en/documents/3984345. Accessed: 02-23-21.

[14] Honggfuzz. https://github.com/google/honggfuzz. Accessed: 02-13-21.

[15] Jq. https://github.com/stedolan/jq. Accessed: 02-13-21.

[16] libfuzzer. https://llvm.org/docs/LibFuzzer.html. Accessed: 02-13-21.

[17] Microfocus-fortify. https://www.microfocus.com/de-de/products/static-code-analysis-sast/overview. Accessed: 02-13-21.

[18] Microfocus-fortify license agreement. https://www.microfocus.com/media/documentation/micro_focus_end_user_license_agreement.pdf. Accessed: 02-13-21.

[19] Mofuzz: A fuzzer suite for testing model-driven software engineering tools.

[20] Oclint. http://oclint.org/. Accessed: 02-13-21.

[21] Radamsa. https://gitlab.com/akihe/radamsa. Accessed: 02-13-21.

[22] Suricata. https://suricata-ids.org/. Accessed: 02-13-21.

[23] Tesseract ocr. https://github.com/tesseract-ocr/tesseract. Accessed: 02-13-21.

[24] yaml-cpp. https://github.com/jbeder/yaml-cpp. Accessed: 02-13-21.

[25] Context is king: The developer perspective on the usage of static analysis tools. *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings*, 2018-March:38–49, 2018.

[26] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You get where you're looking for: The impact of information sources on code security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 289–305, 2016.

[27] 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)*, pages 3–8, 2016.

[28] Andrei Arusoaie, Stefan Ciobaca, Vlad Craciun, Dragos Gavrilut, and Dorel Lucanu. A comparison of open-source static analysis tools for vulnerability detection in C/C++ Code. *Proceedings - 2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017*, pages 161–168, 2018.

[29] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1597–1612. IEEE, 2020.

[30] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: Fuzz driver generation at scale. ESEC/FSE 2019, page 975–985, New York, NY, USA, 2019. Association for Computing Machinery.

[31] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. *Proceedings 2020 Network and Distributed System Security Symposium*, 2020.

[32] 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, page 1011–1023, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Alexandra Bugariu and Peter Müller. Automatically testing string solvers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1459–1470, New York, NY, USA, 2020. Association for Computing Machinery.

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

[35] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. SAVIOR: towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1580–1596. IEEE, 2020.

[36] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1967–1983, Santa Clara, CA, August 2019. USENIX Association.

[37] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 332–343. ACM, 2016.

[38] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 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*, pages 1497–1511. IEEE, 2020.

[39] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: Large-Scale Automated Vulnerability Addition. *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, pages 110–121, 2016.

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

[41] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2523–2540. USENIX Association, August 2020.

[42] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2577–2594. USENIX Association, August 2020.

[43] Xiang Gao, Ripon K. Saha, Mukul R. Prasad, and Abhik Roychoudhury. 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, page 1147–1158, New York, NY, USA, 2020. Association for Computing Machinery.

[44] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 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*, pages 1613–1627. IEEE, 2020.

[45] Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts. Standard, ISO/TC 159/SC 4 Ergonomics of human-system interaction, March 2018.

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

[47] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Fuzzing error handling code using context-sensitive software fault injection. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2595–2612. USENIX Association, August 2020.

[48] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681, 2013.

[49] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 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.

[50] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A neural network language model-guided javascript engine fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2613–2630. USENIX Association, August 2020.

[51] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers, 2020.

[52] Daniel Liew, Cristian Cadar, Alastair F. Donaldson, and J. Ryan Stinnett. 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, page 521–532, New York, NY, USA, 2019. Association for Computing Machinery.

[53] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. FANS: Fuzzing android native system services via automated interface analysis. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 307–323. USENIX Association, August 2020.

[54] 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, page 1024–1036, New York, NY, USA, 2020. Association for Computing Machinery.

[55] Valentin Manès, Marcel Boehme, and Sang Kil Cha. Fse2020 - boosting fuzzer efficiency an information-theoretic perspective, Jun 2020.

[56] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel von Zezschwitz, and Matthew Smith. "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, page 1–12, New York, NY, USA, 2019. Association for Computing Machinery.

[57] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith. Deception task design in developer password studies: Exploring a student sample. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 297–313, Baltimore, MD, August 2018. USENIX Association.

[58] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. Sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 778–788, New York, NY, USA, 2020. Association for Computing Machinery.

[59] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[60] Yannic Noller, Corina S. Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. Hydiff: Hybrid differential software analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1273–1285, New York, NY, USA, 2020. Association for Computing Machinery.

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

[62] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing.

[63] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided grey-box fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306. USENIX Association, August 2020.

[64] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1629–1642. IEEE, 2020.

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

[66] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. *CoRR*, abs/1708.08437, 2017.

[67] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 19–36. USENIX Association, 2020.

[68] C. Sadowski, J. Van Gogh, C. Jaspan, E. Soderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 598–608, 2015.

[69] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 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.

[70] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. Test suites for benchmarks of static analysis tools. *2015 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW 2015*, (November):12–15, 2016.

[71] Dag Sjøberg, Bente Anda, Erik Arisholm, Tore Dybå, Magne Jørgensen, Amela Karahasanovic, Espen Koren, and Marek Vokác. Conducting realistic experiments in software engineering. pages 17 – 26, 02 2002.

[72] 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)*, pages 221–238. USENIX Association, August 2020.

[73] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bei-Tseng Chu, and Heather Richter. How developers diagnose potential security vulnerabilities with a static analysis tool. *IEEE Transactions on Software Engineering*, PP:1–1, 02 2018.

[74] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 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, page 248–259, New York, NY, USA, 2015. Association for Computing Machinery.

[75] David R. Thomas. A general inductive approach for analyzing qualitative evaluation data. *American Journal of Evaluation*, pages 237–246.

[76] Daniel Votipka, Michelle L Mazurek, Hongyi Hu, and Bryan Eastes. Toward a Field Study on the Impact of Hacking Competitions on Secure Development. 2018.

[77] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. *Proceedings - IEEE Symposium on Security and Privacy*, 2018-May:374–391, 2018.

[78] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 999–1010, New York, NY, USA, 2020. Association for Computing Machinery.

[79] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 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.

[80] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 765–777, New York, NY, USA, 2020. Association for Computing Machinery.

[81] 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, page 789–800, New York, NY, USA, 2020. Association for Computing Machinery.

[82] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1643–1660. IEEE, 2020.

[83] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2307–2324. USENIX Association, August 2020.

[84] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. 2020.

[85] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. Squirrel: Testing database management systems with language validity and coverage feedback, 2020.

[86] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling.

[87] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2255–2269. USENIX Association, August 2020.

A Semi-Structured Interview

A.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.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?

B 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 5: Overview of GitHub projects and reports of Clang Static Analyzer

C 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 6: 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 7: 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 8: 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 9: 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 10: CTF overall

D 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 11: 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 12: CTF Comments and usage in future of the static and dynamic analysis tools

E Demographics

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 13: CS Participant Demographics

Gender	Male: 8	Female: 0	Other: 0
Age	min: 19, max: 32	mean: 23.25, median: 22	sd=4.2, NA=0

Table 14: CTF Participant Demographics