The future of grey-box fuzzing

Isak Hjelt
Abstract

Society are becoming more dependent on software, and more artifacts are being connected to the Internet each day[31]. This makes the work of tracking down vulnerabilities in software a moral obligation for software developers. Since manual testing is expensive[7], automated bug finding techniques are attractive within the quality assurance field, since it can save companies a lot of money.

This thesis summarizes the research of an automated bug finding technique called grey-box fuzzing, with the goal of saying something about its future. Grey-box fuzzing is a breed of fuzzing, where the basic concept of fuzzing is to provide random data as input to an application in order to test it for bugs. To portray the current state of grey-box fuzzing, two tools which are relevant to the current research will be presented and discussed. A definition of what grey-box fuzzing is will also be extracted from the research papers by looking at what they all have in common.

The combination of fuzzing with symbolic execution or dynamic taint analysis are two of the approaches which this work has identified and discussed, but argues that dynamic taint analysis is more promising to the future. Lastly, the trend within fuzzing is predicted to go more towards the grey-box style of fuzzing, which leads to grey-box fuzzing rising in popularity.
Acknowledgements

I would like to thank my supervisor Pedher Johansson for helping me guide this thesis in the right direction and always answering all my questions to the best of his ability. I also want to thank my sister for proofreading this thesis and giving me tips on how to improve my writing.

Last, but by no means least, thanks to my significant other for always motivating and supporting me during my three years of study.
Contents

1 Introduction 1
   1.1 Purpose of the thesis 3

2 The basics of fuzzing 4
   2.1 Black-box fuzzing 4
   2.2 White-box fuzzing 4

3 Grey-box fuzzing 6
   3.1 Strengths and weaknesses 7

4 Grey-box fuzzers 7
   4.1 AFL 7
   4.2 VUzzer 8

5 Grey-box fuzzing now and in the future 8
   5.1 Combining techniques 8
   5.2 Dynamic taint analysis 10

6 Discussion and conclusion 11
   6.1 Conclusion regarding the future 11
   6.2 Conclusion validity 11
   6.3 Importance of categorization 11
   6.4 When should grey-box fuzzing be used? 12

References 13
1 Introduction

With a society that becomes increasingly dependent on technology, finding and eliminating bugs in software is essential. Many people depend on software, the software need to be available when they need it and in a way it is expected to work. Security bugs in software can have devastating impacts on companies and might cost them enormous amount of money [47]. To be able to stand up to the competition to other companies nowadays you will most likely be forced to rely on software whether you like it or not. The research for better software testing methods can not stop, it would be morally wrong if researchers stopped trying to come up with new smart techniques to find bugs in code.

A lot of time and money are spent on testing, often 50% of a software project’s expenses goes towards manual testing [7]. Creating tests manually is expensive, error-prone and most of the time inconclusive [7]. Vulnerabilities in software are often caused by bugs in the code that escape detection of software quality assurance. Despite the efforts of trying to make software more resistant against security vulnerabilities, research from recent years suggests that the vulnerabilities in software are more common than ever [35]. This is a big problem to fight and the internet of things [31] phenomena does not make it easier or any less important.

Bugs are commonly hidden in paths of the code that rarely gets executed. Error-prone software such as parsers or decoders that handles complex file formats must be able to handle many different inputs and corner cases. Bugs in software that handle files or any other complex input can have serious consequences which may lead to security exploits [10]. Sometimes companies purposely ignore their responsibility to test their applications for these security faults and go for a less demanding approach [33]. Because of the security testing problems stated, automated security testing has become more popular. One complementing testing technique that has emerged is called fuzzing. Fuzzing can be used to test applications where the space of possible inputs is large. The technique is used to see how well an application handles unexpected inputs and thereby reveal bugs.

Fuzzing generates input in a random fashion and feeds it to the program. The input might generate an exception, make the program do something valid or even put the program in an invalid state that was never thought of, thus reveal a bug. Generating input for a program is mainly done in two ways: using grammar-based fuzzing which uses a model or format to generate input, or mutational fuzzing which starts with a valid input and changes it in a random way. Grey-box fuzzing is a subject without an excessive amount of documentation, which is partially the motivation for this study being made. It can briefly be explained as a fuzzing technique located between black- and white-box fuzzing, where black-box means only looking at the input/output of the system being tested, and white-box indicates the access to source code. Grey-box fuzzers does not require source code, but are still more refined than a black-box fuzzer since it can still glean information regarding the internal state of the system being tested, usually using dynamic or static analysis on a binary level [27].

Hackers often exploit bugs in software for their attacks, and they mainly use two methods to find them [17]. One of them is by using reverse engineering on binaries to retrieve the original source code (as close as possible). Once the source code is retrieved, it can be inspected with the intention of finding any security flaws. The other method is to use black-box fuzzing on the software to reveal bugs. This method is fruitful for a hackers, since the
process of feeding a system with random permutations of data can easily be automated and run until a bug occurs. Finding one bug in a software is easy compared to finding all of them, even an unsophisticated fuzzer might do the job, but finding one bug might be all it takes for a malicious hacker to do damage. To fight this problem, ethical security specialists need more sophisticated fuzzing methods to find and fix bugs before they can be found and exploited by hackers.
1.1 Purpose of the thesis

There exists no generally accepted method to distinguish grey-box fuzzing from the other two fuzzing techniques, white-box and black-box [20]. The purpose of this thesis is to clarify the term grey-box fuzzing and analyze the tools which employ the technique, thus give an understanding in where the grey-box fuzzing research stands today and where it is heading.

With the hope of stating something about the future of grey-box fuzzing, these are the aspects that has been focused on:

- Strategies the recent papers are presenting, and how they compare to other approaches.
- What the current state is like within the research of the techniques that grey-box fuzzers utilize.
- Things that are considered to be obstacles within the field of research.
The basics of fuzzing

The idea of fuzzing was first described by Miller in 1989 [21]. The basic idea of fuzzing is to use random strings as input to a monitored software with the intention to uncover bugs. Fuzzing is an automated or semi-automated technique where the input can be based on knowledge of the program internals, totally random, or based on some kind of initial seed. It is typically used to test applications that take structured files as input but might also be used for other things such as network protocols.

The term is another word for *interface robustness testing* [14], where the interface is the attack surface and usually the thing available to the users. Fuzzing is a type of security testing but should not be confused with penetration testing. This technique is used to test how well a system handles unexpected input, usually with the intention of finding memory related errors like buffer overflows, heap overflows, stack overflows and the like, fuzzing might trigger any kind of bug, but deciding when a bug has occurred is a job for the test oracle [2].

Determining when a bug has occurred is a hard problem that requires an oracle to know what behavior to expect given a certain input. If a fuzzer manages to trigger a bug, but nothing registers its occurrence, the work of triggering the bug is insignificant. Research in the fuzzing technique has enhanced the knowledge of random input generation, but fuzzers still inhabits a bottleneck from the *test oracle problem* [2]. This topic is of importance for automated testing techniques, and improving it could have a big impact on the future.

2.1 Black-box fuzzing

Black-box fuzzing is what we usually call traditional fuzzing. This is the most simple form of fuzzing and is based on the assumption that the input and output of the SUT (System Under Test) is the only thing known to the fuzzer. The inner workings of the SUT is unknown, therefore making it a black-box. An example of this would be a network protocol. Both the server-side and client-sides implementations could be fuzzed for vulnerabilities [38]. As mentioned in the Introduction, an attacker could leave the fuzzing process running until a bug is revealed. Since implementations of protocols might be the same on many servers, an attacker could set up the same system to run the fuzzing process against. If a vulnerability is discovered, it can in theory, be exploited on every server running the same implementation of that protocol.

2.2 White-box fuzzing

Unlike black-box fuzzing, white-box fuzzing uses program analysis to understand the impact of the input and increase code coverage of the SUT. White-box fuzzing takes advantage of its access to the source code and design specifications of the SUT. Symbolic execution is highly associated with white-box fuzzing and is a way of determining how inputs causes different paths in the program to execute. During symbolic execution, input variables take symbolic values instead of concrete values. All the actions done to the symbolic value will be stored, and later taken into account when reaching an if statement. This enables the symbolic execution to set the symbolic value so that it will take a certain path with consideration to the actions done to the symbolic value. Constraints can be gathered during symbolic executing of a program. The constraints are gathered from conditional branches encountered...
along the execution, negated, and solved using a constraint solver. The solutions provided by the solver are then used in the generation of new inputs. The inputs are then used to discover new paths or reveal security vulnerabilities. White-box fuzzing was used in the development of Windows 7 and discovered one-third of all the vulnerabilities found prior to the release [6].
3 Grey-box fuzzing

The term grey-box fuzzing is mentioned for the first time 2007 by Demott, Enbody and Punch [14]. They state that they have created a new "breed" of fuzzing but also mention that they are the only grey-box fuzzer that uses evolutionary computing, which indicates that there is other grey-box fuzzers around at the time of writing. Identifying those other fuzzers is not straightforward, mainly because the documentation of the tools never mention what kind of fuzzing they are utilizing, examples of this is one paper by Cha, Averinos, Rebert, and Brumley, 2012 [10] and another by Wang, Wei, Gu, and Zou, 2010 [40].

Even though there is an insufficient amount of literature defining what grey-box fuzzing is and what it is not, most of the literature concerning grey-box fuzzing presents concrete implementations of fuzzers which they categorize as grey-box fuzzers [14, 5, 27]. There are several articles that touches on the matter, for example one by McNally, Yiu, Grove and Gerhardy, 2012 [20], but the more thorough information can be found in articles presenting an implementation that adopts the grey-box fuzzing technique. By looking at the literature, there are some facts regarding grey-box fuzzing which seems to be agreed upon by the authors. In some papers [26, 20], grey-box fuzzing is implicitly defined as a subcategory to grey-box testing, thus suggesting that grey-box fuzzing is more about what a fuzzer knows about a SUT, rather than what the fuzzer does to the SUT. Delineating between the different techniques is still not an easy task. Since white-box fuzzing is characterized by its access to the source code, what should a fuzzer that uses reverse engineering to first recover the source code of the SUT be called? For example, in the article Model-Based White-box Fuzzing for Program Binaries [25], they present a fuzzer that uses reverse engineering on program binaries to perform lightweight analysis. Should this be called a grey- or white-box fuzzer? It is a grey-box fuzzer since it does not require access to the source code, and it analyses more than the in- and output of the program. But it is still a white-box fuzzer seeing that it has access to the source code, even though it did not initially.

Now that grey-box fuzzing is said to be situated in between black- and white-box fuzzing, does it in practice manage to capture the benefits of the respective technique? To determine if the tools capture the strengths and minimize the weakness of black- and white-box fuzzing, these attributes needs to be clarified. Starting with black-box fuzzing, as it is the foundation of fuzzing. The technique is still used in a variety of fuzzing tools and frameworks, such as, Peach [24] and Sulley [37]. So, why is this simple technique which has been around since 1990 still used when there are more sophisticated techniques around, like white-box fuzzing? It is simple to write a custom black-box fuzzer and anyone can do it in a limited amount of time [16]. Despite being bound to provide input and look at the output of the SUT, black-box fuzzers can still be clever in the way they generate inputs, and therefore achieve high code coverage without suffering from problems in scalability [18, 27]. Although black-box fuzzing can achieve high code coverage, its limitations are still a fact, and with no information about the inner workings of the SUT, the chances of exercising a certain path can be slim. As explained in [6] with the help of the following code example,

```c
int foo(int x) { // x is an input
    int y = x + 3;
    if (y == 13) abort(); // error
    return 0;
}
```
there are times when a path has a low chance of being exercised by randomly mutating well-formed input. Assuming that the integer is a 32-bit value, there is only a $1 \in 2^{32} - 3$ chance of exercising the then branch of the conditional expression. One of white-box fuzzing's capabilities is to cope with this sort of limitation. White-box fuzzing solves this problem using symbolic execution, which means that by inspecting the code during execution, constraints from conditional statements can be gathered, and be used to determine how the input should be generated to exercise new paths. Grey-box fuzzers can also solve problems like this using symbolic execution, but are also capable of doing it using dynamic taint analysis.

3.1 Strengths and weaknesses

The main strength of grey-box fuzzing is that it does not require the source code to perform the fuzzing. This is beneficial for testing and security verification of third-party software [14]. Since grey-box fuzzing is based on the lightweight black-box fuzzing technique, but may still glean information about the SUT, code coverage can still be leveraged without sacrificing time on program analysis [5]. Grey-box fuzzing is however not suitable for situations when the internals of the program cannot be accessed for different reasons.

4 Grey-box fuzzers

This section focuses on describing two on the current tools which are interesting to the research of grey-box fuzzing. The purpose of this section is to give concrete examples of the grey-box fuzzing technique being used in software testing tools. AFL and VUzzer were selected as the tools to be presented in this section. AFL was selected based on its connection to field and its repeated appearance within the research papers [35, 5, 27, 32]. VUzzer was mainly chosen since the article presenting the tool was published in 2017 (Same year as this thesis was written), and also because it tries to target some of the flaws that AFL possesses. Other similar tools exist, but are not always explicitly classified as grey-box fuzzers. Some of the other tools employing this technique is EFS[14], Driller[35] and LynxFuzzer[19]. An article presenting EFS was published 2007, which is the first article about grey-box fuzzing that has been identified during this thesis.

4.1 AFL

AFL (American Fuzzy Lop) [44] is a state-of-the-art [5, 27] coverage based grey-box fuzzer. It is responsible for uncovering many vulnerabilities in lots of popular software, such as, bash, OpenSSL and Mozilla Firefox [45]. Since the launch of AFL, many improvements have been made to the tool. Some of the improvements made to AFL have been due to scientists employing their fuzzing solution into AFL [35, 5], this gives an indication that AFL is considered one of, or even, the leading grey-box fuzzer in the area of research.

AFL uses an evolutionary algorithm to find new test data that can be used as input to the SUT. The input generation is done using a feedback loop which determines how good an input is. Any input that exercises a new path in the program is considered good and therefore retained for mutation. The data gets mutated and thereafter used as input to the SUT to see if it leads to a new path. AFL is considered application-agnostic [27] and does not know how to mutate the input in a beneficial matter (i.e, so that it explores the most meaningful
It is also worth mentioning that AFL can inject instrumentation code straight into the binary or at compile time, which is an important ability to not be dependent on source code.

4.2 VUzzer

VUzzer\[27\] is an application aware grey-box fuzzer that targets some of the flaws that AFL has but does also preserve its strengths. One of the weaknesses that the researchers behind VUzzer has targeted to solve is AFL’s mutation strategy. Like EFS \[14\], VUzzer utilize an evolutionary fuzzing strategy, an evolutionary process will be used for input generation until a crash occurs, or the maximum number of generations is reached.

VUzzer focuses on generating meaningful input that explores new interesting paths in the SUT. The term input gain (IG) is used by the researchers of VUzzer; IG indicates an input’s ability to exercise new paths. VUzzer strives at generating non-zero IG by answering the question: Where in the input to mutate and what value should be put there?

By static and dynamic analysis of the SUT VUzzer creates a ”smart” feedback loop. What the researchers mean with the loop being smart is that it considers control- and data-flow features from the previous execution and uses that information to generate new input. The data-flow features presents information about the association between computations within the SUT, and the input data. By extracting this information, VUzzer can solve the question about where and how to mutate. The control-flow feature allows reasoning about the significance of an execution path. Usually error handling blocks are not of importance and can be ignored, thus speed up the generation of input which will take the fuzzing to more relevant parts of the SUT. Other than ignoring error-handling blocks, VUzzer can prioritize paths that are more likely to take the execution deeper into the program.

5 Grey-box fuzzing now and in the future

5.1 Combining techniques

Unlike the classic grey-box fuzzing tools discussed earlier, there are other promising approaches that has recently emerged, which combines grey-box fuzzing with other techniques. These techniques are combined in an attempt to tackle the weaknesses that fuzzing has. In 2016, a paper \[35\] presenting a tool called Driller was published. The tool tries to mitigate the path explosion problem, and the need for ”input test cases”, by combining concolic execution(also called dynamic symbolic execution) with fuzzing. The path explosion problem, where the number of paths exponentially grows for each branch and becomes unmanageable, is something that concolic execution is considered to suffer from \[32\]. They leverage AFL as the fuzzer to combine with their approach, making it a grey-box fuzzer. In the article, Driller gets categorized as a white-box fuzzer, but since they explicitly state that they ”opted for a QEMU-backend to remove reliance on source code availability” it should, in my opinion, be categorized as a grey-box fuzzer. In the paper they motivate their research by stating that fuzzing is proficient at providing general values within a compartment, but struggles at the more precise values, with the consequence of struggling to take the execution between compartments. On the contrary, concolic execution is clever when it comes to
selecting the right value to pass specific checks. Combining these two strategies results in a fuzzer that is able to solve branch constraints in a sophisticated matter, thereby take the execution to deeper path explorations.

Similar approaches like Driller has been presented before, for example in this master thesis [23]. In the thesis, the concept of what they call hybrid fuzzing is examined. Hybrid fuzzing is explained as the combination of symbolic execution and classic black-box fuzzing. It is stated that the combination of techniques is based on their different characteristics, one being faster at exploring deeper in the code, and the other being better at achieving code coverage in breadth. The method initially uses symbolic execution to identify "frontier nodes", which ensures that different paths are taken early in the execution. The Driller article states that a problem with this method is its inability to solve complex checks deeper in the SUT, which is not a problem for Driller.

The previously discussed articles [35, 23] are similar in how they combine fuzzing with symbolic execution, both mention being hybrid, and that they combine the best of both worlds. Other fuzzing tools does also combine regular fuzzing with other techniques, but are still not called a "hybrid fuzzer". To make advances within the fuzzing research, combination of techniques is unavoidable, thereby forces the fuzzing term to broaden its meaning. VUzzer, and SAGE are two clear examples of this phenomena, none of them are using pure random fuzzing as an underlying method, the fuzzing connection might even be hard to spot, but does still categorize themselves as fuzzers [17, 27].

However, in the recent paper regarding VUzzer [27], they argue that heavyweight and non-scalable solutions like Driller and similar is not the definitive solution. They argue that the key strengths of fuzzing are its ability to be lightweight and scale well with bigger programs, that reasoning does not however exclude that there are other techniques which could be interesting to combine with fuzzing. The article also mention that VUzzer could be improved if it was combined with other techniques. An example of this is the method presented in [42], which focus on how to schedule the fuzzing of a given set of seed pairs, so that it maximizes the number of found bugs at any point in time. This method is very similar to the work presented in [28, 11], which are all recent studies, focusing on optimizing seed selection and choice of mutation ratio.

Although the VUzzer paper dismisses symbolic execution as a way forward for fuzzing, by stating that it is hard to scale, suffers from the path explosion problem, and therefore it weakens one fuzzing’s key strengths, there are papers that present solutions to fight those weaknesses. This [1] article presents a tool called Veritesting, and shows how it enables symbolic execution for large-scale bug finding. On the following code,

```c
int counter = 0, values = 0;
for ( i = 0 ; i < 100 ; i ++ )
    if ( input [i] == 'B' ) 
    {
        counter ++;
        values += 2;
    }
if ( counter == 75) bug ( ) ;
```

which has $2^{100}$ possible paths, Veritesting was able to find the bug and achieve full code coverage within 47 seconds, compared to four other state of the art symbolic executors,
where none of them managed to find the bug within an hour. Symbolic execution suffering from the path explosion problem is stated as a fact throughout the grey-box related literature discussed in this thesis, which might be the case right now. To state that symbolic execution is not scalable, and therefore not the answer for fuzzing can on the other side be dangerous to the field of research. As shown above, and presented in this paper [9], there are solutions to make symbolic execution scale better. In this [9] article the author explains the technique of combining Heuristic search with symbolic execution, and states that it has shown promising results which could lead to groundbreaking impact on the path explosion problem.

Given that there are many papers presenting new improvements to the fuzzing research, and there are state-of-the-art fuzzers which are still not implementing those techniques, modern grey-box fuzzers could benefit a lot from implementing already researched improvements within the field of fuzzing. In order to take advantage of the potential improvements available when combining techniques with gray-box fuzzing, it is important to keep an open mind and not get stuck on the fact that some weaknesses of certain techniques will remain. It is considered a fact that symbolic execution is not scalable, which can be seen in a lot of literature and publications, not to mention Vuzzer’s speech about it not being the right way forward. Scientists within the field needs to constantly keep themselves updated on progress regarding previously tested techniques, so that they can benefit from possibly new solutions.

5.2 Dynamic taint analysis

Grey-box fuzzers employs lightweight program analysis in some way or the other, this is mainly monitoring of the SUT [27]. Some fuzzers are more application-aware than others, symbolic execution is one way to enable reasoning about a software under execution. Another method that has been used in grey-box fuzzers is dynamic taint analysis [27, 40]. Dynamic taint analysis can during the execution, tell which memory locations and registers relate to tainted inputs, thus map the values to their related offset in the input. Values in the program that originate or is arithmetically derived from a taint source is considered tainted values [29]. Taint sources are where tainted inputs are derived, for example, network packets, files and user input [27]. The purpose of the technique is to track the tainted input data within the application, from source to sink (where the data comes from and where it ends). The technique is commonly used to tie untrusted input to an exploited vulnerability. By tracking the propagation of tainted data, dangerous behavior like overwrite attacks and such, caused by tainted input can be detected. This enables an easy way to find relations between which part of the payload that led to the exploit [22].

This technique suits the grey-box fuzzing approach well, and one of the reasons is that it does not require any source code or special compilation instrumentation [22]. Another reason why dynamic taint analysis goes well with fuzzing is its core ability to trace tainted input to events within the SUT. Those are the properties that VUzzer take advantage of to trace which offsets of the input that taints operands at cmp instructions. Dynamic taint analysis can be leveraged at cmp op1, op2 instructions to determine how the operands are tainted by a set of offsets. This is done at byte level, and can for each byte in the operand provide information about at what offset of the input that taints a specific byte.

Dynamic taint analysis has been used in many different security research fields beyond grey-box fuzzing and the number of applications which utilize the technique is enormous [29]. The technique is used to check whether user input is executed, thus prevent code injections
6 Discussion and conclusion

6.1 Conclusion regarding the future

As mentioned earlier, symbolic execution is considered to suffer from the path explosion problem, thus is hard to scale with bigger systems. Dynamic taint analysis on the other hand, is scalable, but most importantly used in many other related research areas. The technique might not be better than symbolic execution if the path explosion problem was mitigated, but the research already done on dynamic taint analysis could be a big advantageous factor, hence be more successful within the grey-box fuzzing field.

Other than which technique that looks most promising for grey-box fuzzers, I agree with McNally [20] when he states that fuzzing is trending more towards the grey-box style than the other two. He argues that, the trend is a result of the increasing availability of tools to perform binary analysis, which I think is one of the contributing factors. I think the main reason however, is the availability of open-source state-of-the-art tools like AFL [44] and VUzzer [27], which allows for research approaches to be combined, compared and analyzed using these tools; therefore leads to more progress within the field. This could be a problem for the white-box fuzzing research, since the most established white-box fuzzer within the research, SAGE [17], is not publicly available and only used within Microsoft. Black-box fuzzing will probably still be around since it fills another purpose, like testing systems without access to the binary file or the source code. Frameworks, like Peach [24] which enables faster creation of custom-made black-box fuzzers for particular scenarios can also be attractive in some cases.

6.2 Conclusion validity

Making assumptions about whether a complex technique is more promising than the other can be difficult if you are not an expert in the area. The assumptions made in this thesis are based on statements from articles related to the field, but also based how widely adopted the each technique is within related security testing areas. The future of grey-box fuzzing is also affected by other things such as the demand and availability of tools that employ the technique, where the availability is the main factor discussed in the conclusion.

6.3 Importance of categorization

You might wonder if categorizing fuzzers is of any value, if mentioning whether the fuzzer is a black, white or grey is important, and if anyone benefits from this information. First, I can speak from my own personal experience and say that being new to the concept of fuzzing and not understanding those categories made it hard to understand why articles kept bringing them up. Intuitively I think that there must be a reason to why those terms are so
widely used in the literature. Some papers even have the classification of the fuzzer in the title, unlike some, which only implicitly provides this information, leaving it to the reader to classify the fuzzer for themselves.

From a company’s point of view, I think that distinct classification of fuzzers can be helpful when trying to identify fuzzers suitable for their needs. Categorization provides guidance for people trying to find fuzzers best suited for a specific application where there might be limitations concerning access to source code or other internal knowledge about the application. Sometimes companies might want to separate testing from development, hence it might be useful to know which fuzzers to even consider looking at.

### 6.4 When should grey-box fuzzing be used?

Information regarding when grey-box fuzzing is more suitable than black- or white-box fuzzing can be interesting for anyone interested in running this technique against their system. Since one of the key properties of grey-box fuzzing is that it does not require any source code, it might seem like a technique only directed towards black-hat hackers. Some of the arguments for using grey-box fuzzing is the increasing availability of tools to perform binary analysis [20], examining the binary provides the “ground truth” since computers execute binaries [34], and the scalability being better compared to symbolic execution (usually used in white-box fuzzing) [27]. Shoshitaishvili et al. 2016 [32] points out that the importance of binary analysis is on the rise, with the motivation of it being the only way to prove or disprove properties of a program, properties gathered about a program from analyzing the source code might not hold after compilation [39]. This fact might be worth considering when choosing between different breeds of fuzzers. Even when the source code of a system is available, grey-box fuzzers enables the SUT to be analyzed in a more honest fashion which represents the potentially vulnerable system more accurately.

As mentioned in the introduction, automated software testing techniques are attractive, and software might not be tested properly if the testing process takes too much time. Grey-box fuzzers can be more generic than white-box fuzzers when analyzing applications. Since white-box fuzzers analyze the internals of a program using the source code, knowledge about the languages syntax is required in order to perform analysis. This can make it hard to automatically test software which are a combination of different languages and will also hinder the fuzzing tool to be used on any language that is not known by the tool.

Grey-box fuzzers can unlike black-box fuzzers automatically learn to create valid inputs [46] which removes the need for seed inputs. By not being dependent on valid seed inputs that have the purpose of taking the execution away from branches that only lead to exception handling in the parser code, grey-box fuzzers can be automated in a more sophisticated matter than black-box fuzzers.
References


