DocFuzz: Using Documentation to Produce Better Fuzzing Stubs

Capooci, Corey
cvcapooc@ncsu.edu

May 7, 2021

Abstract

Fuzzing is a testing technique that involves invoking a software's API with carefully curated inputs. Fuzzing discovers vulnerabilities by covering as much code as possible with a wide variety of inputs in order to find vulnerabilities. The latest in fuzzing technology must lower its implementation costs and improve its code coverage before it becomes widely adopted. Our solution, DocFuzz, attempts to eliminate the tedious manual task of writing fuzzing stubs while improving code coverage by leveraging documentation. DocFuzz solves these issues by inputting a library's documentation into a natural language processing (NLP) neural network model to generate a fuzzing stub. We tested DocFuzz against FuzzGen to understand the performance and usability improvements gained from using the NLP model [20]. This paper contains an analysis of FuzzGen and describes the usability and performance issues associated with it. The scope of this paper only studies the first steps of fuzzing stub generation, the generation of the library's API. During these experiments, DocFuzz did not generate any API functions for the libraries that were tested. FuzzGen took longer to generate its API, but discovered over 100 API functions for all of the libraries. These results show that DocFuzz's techniques provided no value in one of the first steps of fuzzing stub generation.

1 Introduction

Fuzzing is a technique used to discover errors in third-party software. Fuzzing attempts to discover errors by using carefully crafted inputs. The fuzzing engine chooses inputs to execute the largest percentage of code possible. Fuzzing engines utilize various techniques to determine the inputs, from the mutation strategies of the American Fuzzy Lop (AFL) to dataflow analysis of GREYONE [1, 16]. These techniques discover vulnerabilities by targeting code that is less frequently utilized and by exercising a wide variety of inputs on all of the code. Fuzzing holds a lot of promise for grey-box testing, but requires improvements to implementation costs.

Fuzzing requires tedious handwritten stubs that hinder the cost effectiveness of the testing, but automatic stub generation looks to solve this issue. Users of fuzz testing write stubs for fuzzing engines to execute. Manually writing the stubs requires expertise in both the fuzzing engine and the library under test, and it is a tedious process to generate this stub. Automatic stub generation solves this issue by creating a fuzzing stub with the help of information gathered from the programs that use the library, the library's binaries, and its documentation. The generation of the stub increases the efficiency of fuzzing by reducing the need for esoteric knowledge and manual labor. In addition, the automatically generated stub can improve code coverage when compared to stubs generated by experts [20]. Current stub generation technology uses techniques such as whole system analysis, or static analysis, to generate the stubs.

FuzzGen and FUDGE use algorithms that may be too slow or impractical for the common tester [20, 9]. FuzzGen and FUDGE rely on long-running static analysis. These applications utilize existing programs and libraries to understand how the library under test is used. The applications generate a fuzzing stub according to the analysis. According to [20], the newer FuzzGen achieved about 55% code coverage on average against its test libraries. There is currently no information about the speed of stub generation, but we suspect that due to the type of analysis it uses that it may be a lengthy operation. To improve usability, we introduce DocFuzz in order to shorten the time it takes to generate a stub while improving the code coverage.

Our solution, DocFuzz, uses natural language processing to increase coverage and decrease stub generation times to improve usability. DocFuzz is an NLP neural network model that uses library documentation to generate a fuzzing stub. DocFuzz decreases the stub output time by avoiding the difficult and computationally intensive task of analyzing copious amounts of consumer code for library under test usage. In addition, the library's documentation helps DocFuzz to better progress through all states of the library code, not only those most commonly used. The combination of increased code coverage and decreased generation times helps to make fuzzing more

cost effective. This paper focuses on the first steps of fuzzing stub generation, the process of inferring the library's API. We compare DocFuzz's API list against FuzzGen's.

In this paper, we test DocFuzz against FuzzGen to understand the benefits of using natural language processing to generate the information necessary to create fuzzing stubs. We compare DocFuzz against FuzzGen in both execution time to generate the API list, and total API found. In addition, we understand the performance of the neural network model by testing the total API found by the model over its maturation. The results of these experiments helps us to gauge the effectiveness of using DocFuzz's approach to inferring API.

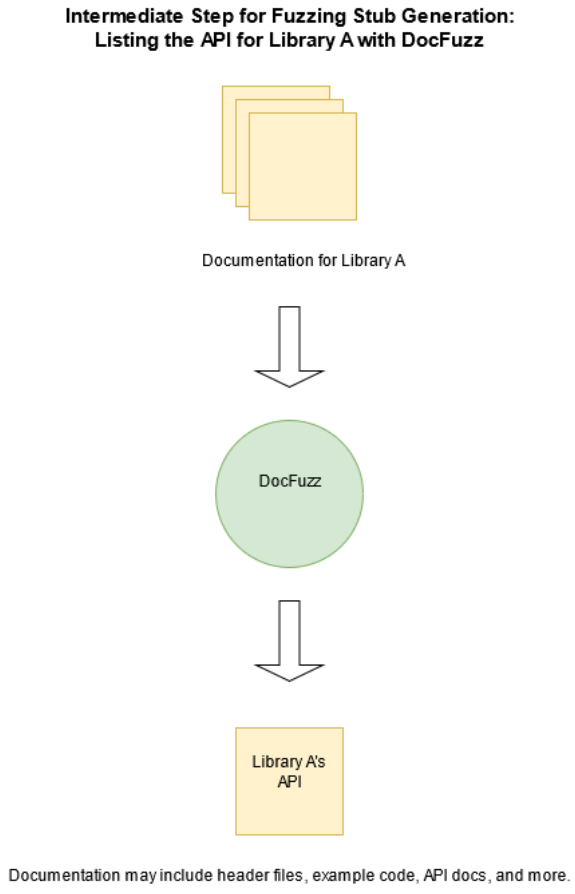

Figure 1: A high-level understanding of DocFuzz as it stands now. It generates a library's API from the libraries documentation.

Our experiments show that DocFuzz completed its analysis in less time than FuzzGen, but was unable to generate a list of API. The results of the experiments show that DocFuzz completed its analysis in under five minutes for all three of the tested libraries, but was not able to infer any API from the documentation. FuzzGen on the other hand generated a comprehensive list of API, and it took as long as 15.5 hours to complete its analysis. Since DocFuzz was unable to produce any API, the maturation of the model did not show any improvements with more training. The current data indicates that DocFuzz is not mature enough to understand API, and needs more development to at least generate a list of possible APIs. These results show that the simple approach to NLP taken in DocFuzz does not provide any value to fuzzing stub generation.

This paper makes the following contributions:

- **Analyzes the usability of the state of the art fuzzing stub generator.** This paper presents a simple analysis of FuzzGen to understand which aspects of fuzzing stub generation need the most improvement.
- **Presents another technique for generating fuzzing stubs.** We venture away from conducting only static code analysis. This work opens the possibilities to using different technologies to generate fuzzing stubs and shows the benefits and downfalls of using documentation to generate fuzzing stubs.
- **Evaluates the performance of fuzzing stub generator by execution time.** The time it takes to generate a stub is crucial for the wide adoption of fuzzing. We emphasize this by providing this information as a point of comparison.
- **Models the performance of the NLP model in regards to its maturation.** Our work shows the performance of the current neural network model, and determines if the current model can provide better results after more training.

The remainder of this paper proceeds as follows. Section 2 provides an overview of our approach to investigating this problem. Section 3 describes the design of our solution, the design of our experiments and their results. Section 4 evaluates our solution and interprets the results. Section 5 discusses our assumptions and possibilities for future work. Section 6 describes related works. Section 7 wraps up and provides our final conclusions.

2 Overview

DocFuzz attempts to improve the usability of fuzzing stub generation by analyzing the documentation of libraries with natural language processing and neural networks. This attempts to circumvent the current static analysis techniques, which take many hours to execute and require a lot of disk space. The first step to improve fuzzing stub generation is to generate an accurate API list. DocFuzz trains a neural network model to understand library

API and focuses on training one portion of the processing pipeline. With this technique, DocFuzz attempts to decrease processing times while generating an accurate list of the library's API. Figure 1 shows the current functionality available in DocFuzz.

3 Design

In order to improve code coverage and to increase the usability of fuzzing, we propose creating fuzzing stubs automatically with the help of natural language processing and neural networks. Automatic fuzzing stub generation limits the amount of work to fuzz a target software. In this solution, the neural network model determines the information necessary to generate the fuzzing stub, but in this paper we use the model to accomplish one of the first steps of fuzzing stub generation, the inference of the library's API.

DocFuzz relies on the spaCy natural language processing tool to generate its model, and return the API of the documentation. spaCy's processing pipeline is configurable and contains some default components to help with the language processing. DocFuzz relies primarily on the entity recognizer to find the API within the documentation. Since API entities are not recognized by default in spaCy, DocFuzz needs to train the entity recognizer to understand what an API is. To do this, we created training data from library documentation.

The training data was generated from 3 different libraries, kfrlib, tacopie, and ygg, to train the entity recognizer. The steps to creating the training data include manually determining the API of each of the libraries, choosing which documentation can be used, using spaCy to find the API in the files, and processing the data to create a format that spaCy can then load and process. First, we spent time sifting through documentation and online resources to determine the API of the three libraries. In the end, a library's API was inputted into a single file with one API function on a line. Then, we compiled the documentation into a single folder. All documentation that a developer could use to understand a library was used. This includes header files, man pages, html pages, and more. These files were processed with spaCy to find all instances of the predetermined API. Then, spaCy inserted notations to the end of each document that lists the location of all of the API functions and declaring them as API entities. These files trained the entity recognizer on how to determine API.

The trained model serves as the basis of DocFuzz's API inference code. DocFuzz inserts the trained entity recognizer into its processing pipeline. DocFuzz relies on the performance of the pipeline to generate the API list quickly. As documents are processed, the entity recognizer should pick out the entities that most resemble API and label them as such. At the end of the processing, DocFuzz outputs the list of the API. In the final solution, DocFuzz would use the API list to generate the fuzzing stub.

The rest of this section proceeds as follows. Section 3.1 describes the threat model and why fuzzing is practical. Section 3.2 describes the questions we want to answer with this research. Section 3.3 illustrates our test methodology. Section 3.4 maps our evaluation plan and the results of the experiments.

3.1 Threat Model

In this situation where libraries are open to everyone, the adversary is anyone who intends to understand the vulnerabilities of a library in order to exploit others. The adversary does not need any special privileges. Furthermore, the adversary does not need access to any specific computer or access to any elevated privileges. He or she needs the computing power and the expertise to test a library for vulnerabilities that he or she can exploit.

The goals of the adversary are to discover vulnerabilities in any of the publicly available libraries. Then, he or she wants to exploit these vulnerabilities for her or his gain. The adversary inspects and tests many of the libraries that are freely available and publicly used. Once the adversary discovers a vulnerability in any of the libraries, he or she exploits any software that uses this library to gain information, money, or other profit from these attacks.

In order to succeed in his or her attack, the adversary needs access to the Internet and a personal computer. We know that the adversary can access libraries with vulnerabilities. We expect every library has vulnerabilities. The adversary must be able to test a library effectively enough to discover vulnerabilities that he or she can leverage. We expect that the adversary knows which software these libraries are used in and how to exploit them within the software.

It is not necessary for the adversary to have many privileges or permissions to exploit a library. The adversary does not need explicit permission or access to a certain computing source to discover vulnerabilities in a library. These libraries are open-source and widely available. Therefore, there are few hurdles to discover weaknesses in libraries. The adversary only requires a computer to access the Internet in order to download the libraries and test them for vulnerabilities.

The trusted computing base consists of those who develop the software and fix the bugs. Since these libraries are openly available, everyone else who is using the library cannot be trusted since he or she may use them for nefarious purposes. Only those who develop the software and fix the bugs are trusted since the community relies on them to fix vulnerabilities and produce secure software. If

they cannot be trusted, then the library is never secure and can never be relied upon. Freely available libraries cannot discriminate between users, so all users cannot be trusted. Fuzzing must be used to expose vulnerabilities in order to be fixed before they can be attacked by adversaries.

3.2 Research Questions

This paper attempts to answer these questions.

RQ1: *How usable is the state of the art fuzzing stub generator?*

RQ2: *On average, how long does it take to generate the API from documentation and how does it compare to other methods of API interpretation?*

RQ3: *Can DocFuzz generate an API list that is equivalent or better than the state of the art fuzzing stub generator?*

RQ4: *With every new training set added to the model, can DocFuzz find more API functions? Can we show a trend?*

3.3 Methodology

To determine the effectiveness of DocFuzz, we devise a series of tests and studies to understand its performance. Initially, we analyze the latest in fuzzing stub generation technology, FuzzGen. This provides information on the current implementation costs of fuzzing stub generation. Next, we analyze the performance of DocFuzz by calculating its API output, its execution time to generate an API list, and its performance during training.

To determine how long it takes to generate a stub from documentation, we calculate the length of time for the natural language processing model to output an API list for multiple libraries. We use Linux's time command to calculate the time it takes the application to generate the API list. We compare these execution times of DocFuzz against FuzzGen to understand its performance [20].

To determine how the API list improves with the use of DocFuzz, we compare the outputted API lists of DocFuzz against the lists generated by FuzzGen [20]. We conduct this comparison across multiple libraries. We compare the contents of the lists for similarities and difference, and try to interpret which of the lists is more accurate.

To determine how much better DocFuzz performs with additional training, we track the changes in the API list over the maturation of the application's neural network model for a library. We generate the API for one library after inputting the training data for one documentation. After the lists are generated, they are manually analyzed for correctness and completeness. We can then show the improvement of the model as it matures.

3.4 Evaluation Plan and Results

The design of all the experiments include the use of the same operating system, NLP tool, and test libraries. Every experiment is executed on a machine with Ubuntu 20.04 LTS operating system, 16 cores, 47GB of RAM, and over 500 GB of disk space. The libraries under test include libunwind, libexif, and libxml2. Every experiment utilizes spaCy as its natural language processing tool [3]. spaCy generates the models and serves as the processing pipeline. Most of the experiments are compared against another fuzzer stub generator, FuzzGen [20]. We generate the neural network model for DocFuzz using the documentation as the input, and API functions as the expected output. To generate an API list for a new library, we input the text documentation and the model outputs the API list.

3.4.1 Analysis of FuzzGen

We suspect that for **RQ1** the setup and use of FuzzGen requires resources available to most individual programmers with modest assets, and the time to generate the API list or the fuzzing stubs takes on the order of 2 to 6 hours. The techniques that FuzzGen uses is known to be a lengthy process and it is reasonable to think that it will takes hours to complete this generation.

This experiment requires the resources necessary to build and run FuzzGen. The most current version of FuzzGen requires clang version 6 to compile and a consumer application. For this experiment, we used the Android Open Source Project (AOSP) as the consumer application. FuzzGen was built with the AOSP in mind, so it became the obvious choice. The experiment attempts to understand FuzzGen's usability, its resource usage, and its simplicity of setup.

First, we evaluate the resource usage of this setup of FuzzGen. FuzzGen requires 26 MB. LLVM version 6, which is used to build clang, requires 2.7 GB. AOSP requires at least 400 GB to check out and build the code [2, 4]. In addition, AOSP recommends 16GB of RAM. These resources may be out of reach for developers or companies who want to conduct a minimalist implementation of fuzzing and cannot commit significant resources.

Next, we attempt to understand the ease of use and setup of FuzzGen. In order to generate the necessary bitcode files for FuzzGen, the user must understand which version of AOSP is compatible with FuzzGen. We determined after some experimentation, that version 9 of the AOSP is compatible with the latest version of FuzzGen. This version of AOSP used a version of the clang compiler that generate bitcode files that can be interpreted by FuzzGen. All other versions of the AOSP that we tested, 7, 8, and 11, had clang versions that generated bitcode files that were incompatible with FuzzGen. This is inconvenient for someone who is not familiar with the AOSP.

It takes time and familiarity with the clang compiler to understand the issue and solve it. In addition, newer versions of the AOSP do not have build processes that easily allows developers to generates the necessary bitcode files as documented as FuzzGen's GitHub page.

Lastly, to better understand what results DocFuzz must strive for, FuzzGen generated a fuzzing stub for a single library. We ran FuzzGen to generate a stub for libhevc. AOSP uses libhevc so it is a great first test of the system. This test ran for 3 days, 18 hours, 23 minutes, and 15 seconds before generating a segmentation fault and failing. This illustrates the time commitment it takes to get FuzzGen running correctly as well as the time commitment to generate fuzzing stubs for a single library.

This experiment successfully illustrated the vast of amount of time and effort it takes for FuzzGen to become a solution for fuzzing stub generation. One requires a lot of resources, including time and knowledge, to set up FuzzGen correctly and to use it effectively.

3.4.2 Speed Evaluation

We suspect that for **RQ2** it takes less time to generate the API list with DocFuzz than with FuzzGen [20]. FuzzGen undergoes extensive whole system static analysis of consumers while DocFuzz uses its training to interpret the API.

In addition to the aforementioned design setup, the experiment uses the time command on Ubuntu to understand the length of time it takes to generate the API list and compares these results across multiple libraries. This experiment generates API lists for the three test libraries with both FuzzGen and DocFuzz. The generation of the API list is called as an input to the time command. After the generation of all of the lists is complete, we compare the results of both generators.

The experiment results include the length of time to generate the three lists for each of the test libraries. We compare length of time to generate the API with FuzzGen API list against DocFuzz in order to understand performance differences between FuzzGen and the DocFuzz. We do not compare the performances across libraries. This test is successful if DocFuzz shows a consistent and appreciable performance advantage over the FuzzGen stub.

Time To Generate API List			
	libexif	libunwind	libxml2
FuzzGen	86m 4s	75m 28s	941m 9s
DocFuzz	1m 51s	1m 53s	2m 1s

Table 1: Results of the Speed Evaluation.

The results in Table 1 show that DocFuzz was able to complete its analysis in a faster time than FuzzGen. The API list generation completed in around 2 minutes or under for DocFuzz whereas FuzzGen took as long as 15.5 hours to complete its API generation.

3.4.3 API Discovery

We suspect that for **RQ3** the generated API list between the FuzzGen and DocFuzz will be similar. DocFuzz relies on gathering more direct library usage information from the documentation that may not be as easily understood from extensive static analysis. If given documentation that explicitly states the API list, then DocFuzz will generate a more accurate model of the API in faster time. Although, if the documentation is not accurate, or lacks enough detail to be effective for the neural network model, then the whole system analysis approach of FuzzGen has the upper hand.

This experiment takes the results of the API generation in section 3.4.2 and analyzes them. No extra setup is needed. The results are gathered by counting the number of functions in the list and comparing them between the two generators.

The experiment results contain the average total number of API functions found per library. The experiment then compares the API inferred by the two generators. This experiment is successful if the API list from DocFuzz is comparable to the list from FuzzGen. We believe that the documentation more directly discloses the API and therefore allows the stub to find the API without the overhead of the whole system analysis. We consider it a success if DocFuzz can meet or exceed the completeness of FuzzGen's list.

API List Comparison			
	libexif	libunwind	libxml2
FuzzGen	142	397	1507
DocFuzz	0	0	0

Table 2: Results of the API List Generated. The number indicates the number of API functions found by the generator.

The results in Table 2 show the vast discrepancy in the performance of FuzzGen over DocFuzz. FuzzGen consistently produced a large of list of potential API functions whereas DocFuzz produced none.

3.4.4 Graph of API Performance After Learning

We suspect for **RQ4** that the API list improves significantly after each additional round of training is added to the model, but after enough data points the model benefits less and less from additional information. As seen in

economics, we expect the results of the experiment to represent the law of diminishing returns. We expect to witness substantial improvements for every additional round of training in this experiment, since documentation varies widely in depth and accuracy. We do not expect to see diminishing returns in this experiment since we do not believe we have enough training data to reach that threshold.

In addition to the aforementioned design setup, the experiment creates API lists from DocFuzz for one library across multiple stages of its underlying model's maturation. This experiment generates a API list for the libunwind library after another round of learning. A round of learning can best be described as udpating the neural network model with a new library's documentation with the sufficient API annotations. We generate the API for three rounds of training since we have three sets of training data. This gives us a timeline of the performance of the model.

The experiment is successful if the code coverage improves appreciably and consistently over each iteration of training. If the model continues to perform better and better without any noticeable diminishing returns, then we expect the model to continue to improve as more data points are introduced.

Performance Over Model Maturation			
	Increment 1	Increment 2	Full Training
DocFuzz	0	0	0

Table 3: Results from the Generated API List. The numbers indicate the number of API functions found by the generator.

The results in Table 3 are representative of the experiment conducted in Section 3.4.3. DocFuzz showed no improvement despite the additional learning rounds. Instead, DocFuzz failed to recognize a single API function.

4 Evaluation

The results of the experiments show that DocFuzz did not prove to be an effective tool in generating the API of any of the libraries. They also show that FuzzGen was able to consistently find and generate an extensive list of APIs. Despite the faster processing time, DocFuzz did not recognize and output a single function after analyzing the documentation. DocFuzz's approach to generating the API most likely lacked a more comprehensive processing pipeline that could understand common library documentation.

There may be multiple reasons for the poor performance of DocFuzz. DocFuzz relied heavily on the training of the entity recognizer. The entity recognizer may have lacked the sufficient training for it to truly understand what entity it was searching for. The paper only used documentation from 3 libraries. This leads to a limited set of correct API, and minimal understanding of where they can be found. Another possible issue is that functions in header files, which can be a valuable reference for developers, are typically not natural language, and this prevent the NLP engine from effectively understanding the text.

FuzzGen proved effective in generating the libraries API, but getting FuzzGen to the point of effectively producing API lists proved difficult. FuzzGen discovered 142 API functions for libexif, 397 API functions for libunwind, and 1507 API functions for libxml2. The generation of the API takes under 24 hours for each of the libraries and most of the time it is much faster. Despite these successes, FuzzGen had its downfalls. FuzzGen took over 3 days while attempting to generate a fuzzing stub before running into a segmentation fault. To run FuzzGen with AOSP, like it was initially designed, it takes a lot of resources, possibly more than a company or individual would be willing to spend on a testing framework. It also required very specific versions of the AOSP, and creating bitcode files for the entire repository. For someone who is unfamiliar with the AOSP, these tasks may take hours to complete.

Overall, the results proved that DocFuzz's approach to API generation is not effective and requires a broader more comprehensive technique. Despite its long processing times and large disk requirements, FuzzGen was able to generate a practical list of API functions. DocFuzz executed its operations in minutes, so it can spare to complete more processing in order to get better results. Although, as of now, DocFuzz's techniques are not effective.

5 Discussion

Due to the poor results of the experiments, all of the steps of creating DocFuzz need to be reevaluated. Some of these steps include the breadth and quality of the training data, and the comprehensiveness of the natural language processing pipeline. In addition, to aid in the development of natural language processing fuzzing stub generator, it is critical to have clear truth data in order to understand its performance.

Despite the results of the investigation, natural language processing tools like spaCy are extremely flexible and configurable. Our approach focused on training the entity recognizer to find API functions within library documentation. This approach relied too much on the performance of a single portion of the pipeline. Future approaches may include categorizing the text based on the type of documentation. The pipeline can then react to different formats of documentation. Header files, which

could be extremely helpful for discovering API, could be processed differently and weighed heavier than installation documentation.

Future work may focus on developing pipelines to understand different languages. Header files often provide a mix of natural language in the function commentary and code. Teaching a processing pipeline to recognize and understand the function declarations could allow the application to develop a list of candidate functions for the API list.

This research would have benefited from a more manageable first step in the fuzzing stub generation process. For example, FuzzGen generates a list of functions in the library as its first step of fuzzing stub generation. If DocFuzz took a similar approach, then the process of recognizing all functions could have been easier and may not have needed the training of the entity recognizer. Most functions are not spelled like English words and this could prove helpful in finding the list of functions. After determining all of the functions, DocFuzz could narrow down the options to determine the API.

Truth data for API is not always available. A lot of the benefits of NLP comes from the ability to train the pipeline with correct, robust training data. It is vital to train the data with accurate data and the difficulty of generating good training data may impact the usefulness of this technique. Future work can target the generation of effective truth data for this application.

Continuing this research can help determine whether natural language processing on documentation can serve as a viable solution for fuzzing stub generation.

6 Related Work

Fuzzing research is broad and extensive, and work on fuzzing stub generation can complement and improve other areas of research in fuzzing. Fuzzing research continues to improve since the days of analyzing randomized algorithms for their applicability to fuzz-like testing [8]. Recent work focuses on improving individual styles of fuzzing. Code-coverage fuzzing attempts to use techniques, commonly mutation-based, to test and cover as much code as possible to find vulnerabilities. Direct fuzzing looks more closely at a chosen portion of the software to find vulnerabilities. Hybrid fuzzing builds upon code-coverage fuzzing by utilizing symbolic execution to venture deeper into the code where mutation-based fuzzing has difficulty reaching. Apart from improving the rates of detection, other researchers focus on improving the run-time of fuzzing or creating application-specific fuzzers. Our work attempts to improve upon all of these areas of fuzzing by allowing fuzzers to venture deeper into software and by allieviating some of the manual work of creating stubs.

This work extends previous work done in FuzzGen and FUDGE [9, 20] that automatically generates fuzzing stubs to analyze C and C++ libraries. FuzzGen is the latest implementation which expanded upon the static analysis approach taken with FUDGE by using "whole system anaylsis". Our work attempts to improve upon the static analysis approach by using Natural Language Processing to infer program order from library documentation. Work from this area helps expand the use of fuzzing by removing much of the manual process of generating fuzzing stubs.

The work on automatically generating fuzzing improves upon the significant achievements in fuzzing by providing existing fuzzers with better stubs. Conventional fuzzers implement a mutation strategy to achieve a high level of code coverage. The open-source american fuzzy lop project (AFL) has become a model for many current implementations of mutation-based fuzzing [1]. Researchers constructed many techniques to improve upon the performance of previous state-of-the-art fuzzers such as AFL. Techniques include the prioritization of code coverage, enhancements to mutation strategies, computation of path constraints without symbolic execution, use of program transformation, use of context-free grammars, and implementation of feedback fuzzing [7, 32, 29, 13, 24, 6]. A slightly different take on mutation-based fuzzing uses taint analysis to explore more hard-to-reach code [16, 15]. In addition, other fuzzers avoid the use of program analysis and rely on Markov chains [11]. Lastly, in response to the focus on code coverage, there has also been work that emphasizes the concept of bug coverage over code coverage [28].

In contrast to the broad code-coverage fuzzers, there exists directed fuzzers, which attempt to test code at a location the user desires. Fuzzing stub generators could aid directed fuzzing by providing the stubs with the correct program execution steps for the requested program state. Current directed fuzzers use techniques varying from the extensive static analysis used in Hawkeye [12] to annealing-based power scheduling used in AFLGo [10].

Other work improves fuzzing by combining conventional fuzzing techniques with symbolic execution. This hybrid fuzzing theoretically tests deeper into the program space in order to improve its code coverage. The downsides of this technique are the performance overhead of inefficiently computing symbolic or concolic execution for every path. With this in mind, multiple papers improve this slow running time by using probabilistic path prioritization or artificial intelligence [19, 14, 36].

Since generic fuzzers have shown to have low code coverage [20], some fuzzers target specific applications to improve bug coverage. With the use of the NLP-based fuzzing stub generator, as long as the libraries have documentation, the fuzzing stub is built for the target appli-

cation. Despite the possible benefits of fuzzing stub generation, many fuzzers specifically target software such as DOM, the Linux Kernel, Hypervisor, and the Javascript Engine [33, 22, 30, 23]. In addition, fuzzers have been created to target errors that occur outside user input, for example connection errors [21].

Additional research focuses on improving the execution time of fuzzers. This research understands the benefits of fuzzing, but notes that the performance limitations can be a significant burden on its use. The ProFuzzer understands the inputs that are most important to finding bugs and enhances the mutation strategy to target vulnerabilities [34]. NEUZZ utilizes smoothing techniques that are implemented using neural network models to prevent performance overhead caused by fruitless sequences of random mutations [31]. CollAFL provides more accurate coverage information to prevent path collisions [17]. This is similar to UnTracer which also improves efficiency by limiting the amount of coverage tracing done [27].

Many software engineering research topics utilize natural language processing or artificial intelligence to improve processes and increase efficiency, but natural language processing has not made a significant impact on current fuzzing techniques [5, 18]. Software engineering and security research utilizing NLP include works on investigating the "total recall problem", improving unit test documentation, and modifying directed fuzzing to utilize natural language processing and artificial intelligence [37, 25, 35]. The most exciting progress, in regards to application to fuzzing stub generation, in software engineering research is in areas like API Misuse [26]. This research and our work both gather information from library documents to infer correct usage of libraries. API Misuse attempts to diagnose issues with incorrect usage of APIs, whereas fuzzing stub generation would use this information to generate accurate program steps. The significant overlap between these concepts allows both areas to benefit from the other's progress.

Automatic generation of fuzzing stubs helps further research in fuzzing by leveraging the current state-of-the-art in software engineering research and fuzzing to improve code coverage and make fuzzing more accessible. There has been significant improvements to fuzzing made over the last few years, including work that extends beyond the common code-coverage mutation-based fuzzer. The strides made in areas like API Misuse make the use of NLP in fuzzing stub generation more tenable. With the combination of all of these advancements, we expect fuzzing stub generation, with the help of NLP, to improve fuzzing and to open this tool to a wider audience.

7 Conclusion

The natural language processing technique used in DocFuzz was not successful, but the analysis of FuzzGen shows there is room for improvement in fuzzing stub generation. Relying solely on the effectiveness of one section of the natural language processing pipeline did not provide any value to fuzzing stub generation. Nevertheless, the analysis of FuzzGen shows that fuzzing stub generation is an area that is ripe for innovation, There is much more research to be done, and a more sophisticated implementation of NLP may prove to be the answer.

References

[1] american fuzzy lop (2.52b).

[2] Android Open Source Project Requirements.

[3] spaCy · Industrial-strength Natural Language Processing in Python.

[4] The LLVM Compiler Infrastructure.

[5] V. Arnaoudova, S. Haiduc, A. Marcus, and G. Antoniol. The Use of Text Retrieval and Natural Language Processing in Software Engineering. *Proceedings - International Conference on Software Engineering*, 2:949–950, 2015.

[6] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. 2019.

[7] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence.

[8] S. R. Aziz, T. A. Khan, and A. Nadeem. ¡ Inheritance Metrics Feats In Unsupervised Learning to classify unlabeled datasets and Clusters In Fault Prediction ¿. (November 2012):1–22, 2009.

[9] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang. FUDGE: Fuzz Driver Generation at Scale Google USA. 2019.

[10] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed Greybox Fuzzing. 2017.

[11] M. Bohme, V. T. Pham, and A. Roychoudhury. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2019.

[12] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. page 14.

[13] P. Chen and H. Chen. Angora: Efficient Fuzzing by Principled Search. *Proceedings - IEEE Symposium on Security and Privacy*, 2018-May:711–725, 2018.

[14] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. SAVIOR: Towards bug-driven hybrid testing. *Proceedings - IEEE Symposium on Security and Privacy*, 2020-May:1580–1596, 2020.

[15] P. A. Chen ByteDance Lab, J. Liu, H. Chen, and P. Chen. Matryoshka: Fuzzing Deeply Nested Branches. 2019.

[16] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen. GREYONE: Data Flow Sensitive Fuzzing. Technical report, 2020.

[17] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. CollAFL: Path Sensitive Fuzzing. 2018.

[18] S. Haiduc, V. Arnaoudova, A. Marcus, and G. Antoniol. The use of text retrieval and natural language processing in software engineering. *Proceedings - International Conference on Software Engineering*, pages 898–899, 2016.

[19] J. He ETH Zurich, M. Balunović ETH Zurich, N. Ambroladze ETH Zurich, S. Anodar, ethzch Petar Tsankov ETH Zurich, and M. Vechev ETH Zurich. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts Learning: §5.2.

[20] K. K. Ispoglou, D. Austin, V. Mohan, and M. Payer. FuzzGen: Automatic Fuzzer Generation. Technical report, 2020.

[21] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu. Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection. Technical report, 2020.

[22] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee. HFL: Hybrid Fuzzing on the Linux Kernel. Internet Society, feb 2020.

[23] S. Lee, H. Han, S. K. Cha, and S. Son. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. Technical report, 2020.

[24] C. Lemieux and K. Sen. Fairfuzz: A targeted mutation strategy for increasing Greybox fuzz testing coverage. In *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, volume 18, pages 475–485, New York, NY, USA, sep 2018. Association for Computing Machinery, Inc.

[25] B. Li, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and N. A. Kraft. Automatically Documenting Unit Test Cases. *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, pages 341–352, 2016.

[26] T. Lv, R. Li, Y. Yang, K. Chen, X. Liao, X. Wang, P. Hu, L. Xing, and P.-w. Hu. RTFM! Automatic Assumption Discovery and Verification Derivation from Library Document for API Misuse Detection. 2020.

[27] S. Nagy, V. Tech, and M. Hicks. Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing. Technical report.

[28] S. Österlund, K. Razavi, and H. Bos. ParmeSan: Sanitizer-guided Greybox Fuzzing. Technical report, 2020.

[29] H. Peng, Y. Shoshitaishvili, and M. Payer. T-Fuzz: Fuzzing by Program Transformation. *Proceedings - IEEE Symposium on Security and Privacy*, 2018-May:697–710, 2018.

[30] S. Schumilo, C. Aschermann, A. Abbasi, S. Worner, and T. Holz. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. Internet Society, feb 2020.

[31] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. Technical report.

[32] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. Internet Society, feb 2020.

[33] W. Xu, S. Park, and T. Kim. FreeDom: Engineering a State-of-the-Art DOM Fuzzer.

[34] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang. ProFuzzer: On-the-fly Input Type Probing for Better Zero-day Vulnerability Discovery. Technical report.

[35] Z. Yu and T. Menzies. Total Recall, Language Processing, and Software Engineering. 2018.

[36] L. Zhao, Y. Duan, H. Yin, and J. Xuan. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. Internet Society, mar 2019.

[37] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. Technical report, 2020.