CarpetFuzz: Automatic Program Option Constraint Extraction from Documentation for Fuzzing

Dawei Wang, Ying Li, and Zhiyu Zhang, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China; School of Cyber Security, University of Chinese Academy of Sciences, China;* Kai Chen, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China; School of Cyber Security, University of Chinese Academy of Sciences, China; Beijing Academy of Artificial Intelligence, China*

https://www.usenix.org/conference/usenixsecurity23/presentation/wang-dawei

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

CarpetFuzz: Automatic Program Option Constraint Extraction from Documentation for Fuzzing

Dawei Wang^{1,2}, Ying Li^{1,2}, Zhiyu Zhang^{1,2}, and Kai Chen^{1,2,3*}

^{1}SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
^{2}School of Cyber Security, University of Chinese Academy of Sciences, China
^{3}Beijing Academy of Artificial Intelligence, China
{wangdawei,liying1998,zhangzhiyu1999,chenkai}@iie.ac.cn

Abstract

The large-scale code in software supports the rich and diverse functionalities, and at the same time contains potential vulnerabilities. Fuzzing, as one of the most popular vulnerability detection methods, continues evolving in both industry and academy, aiming to find more vulnerabilities by covering more code. However, we find that even with the state-of-the-art fuzzers, there is still some unexplored code that can only be triggered using a specific combination of program options. Simply mutating the options may generate many invalid combinations due to the lack of consideration of constraints (or called *relationships*) among options. In this paper, we leverage natural language processing (NLP) to automatically extract option descriptions from program documents and analyze the relationship (e.g., conflicts, dependencies) among the options before filtering out invalid combinations and only leaving the valid ones for fuzzing. We implemented a tool called CarpetFuzz and evaluated its performance. The results show that CarpetFuzz accurately extracts the relationships from documents with 96.10% precision and 88.85% recall. Based on these relationships, CarpetFuzz reduced the 67.91% option combinations to be tested. It helps AFL find 45.97% more paths that other fuzzers cannot discover. After analyzing 20 popular open-source programs, CarpetFuzz discovered 57 vulnerabilities, including 43 undisclosed ones. We also successfully obtained CVE IDs for 30 vulnerabilities.

1 Introduction

As software complexity grows, the scale of code in software increases rapidly. For example, according to OpenHub's analysis report [36, 37], lines of code of popular software Apache HTTP Server and MySQL have reached 1.6 million and 2.9 million, respectively. The large-scale code supports the rich and diverse functions of the software, satisfying the various needs of users. However, it also expands the attack surfaces and increases the difficulty of finding potential vulnerabilities, bringing higher security risks and defense costs.

Coverage-guided fuzzing is one of the most successful vulnerability discovery techniques [69], which keeps mutating the input to increase code coverage to trigger a potential security violation in the software (e.g., write access violations). Most previous studies have focused on optimizing strategies in the fuzzing process to increase coverage, such as seed selection [20,41], seed schedule [3,4,40,54,61,67], mutation [8,32], and feedback strategies [1,6,15,16,30,33,44]. These improvements have significantly enhanced the capability of fuzzers to discover vulnerabilities. As of January 2022, more than 36,000 bugs have been discovered by Google's continuous fuzzing service, OSS-Fuzz, alone [19]. However, although the latest fuzzers use different strategies to choose various seeds and mutate inputs and already have a strong program exploration capability, some code remains unexplored.

The main reason is that these latest fuzzers did not fuzz the programs with some specific command-line options. A *command-line option* (or simply *option*) tells the program which operation to be modified. Some options correspond to different program branches, meaning that some code can only be reached by specifying certain options instead of changing the input file. However, in previously exposed vulnerabilities, only small parts of options were specified. For example, *Libtiff's* 103 CVEs from 2014 to 2020 specified only 20 different options, accounting for 9.8% of the total options, implying that many option-dependent codes may remain unexplored. Since the number of combinations may be large, it's unrealistic to iterate over them all. For example, the popular image processing program *ImageMagick* has 242 different options [23], with 7.1×10^{72} possible combinations.

Some researchers [5, 9, 29, 45, 55, 66] attempted to address this limitation by mutating option combinations. However, many mutated combinations may be invalid due to the lack of considering the relationships among options, such as conflicts (i.e., cannot be used together) and dependencies (i.e., must use together). For example, only 11% of the combinations of *openssl-rsa* generated with the mutation algorithm of prior

*Corresponding author.

research are valid. This paper aims to extract relationships among options but it is challenging for the following reasons.

Challenge - C1. Relationships among options are usually declared in natural language in the documentation and can be declared in entirely different ways, greatly increasing identification difficulty. For example, the conflict between the *-a* and *-b* options can be declared in several ways like, *"-a cannot be used with -b," "-a is mutually exclusive with -b,"* and *"no more than one of these options may be given."* Simple approaches like template matching do not work well on this problem. Additionally, some relationships are declared implicitly and can only be identified by comparing multiple sentences. For example, the *tiffcp* document describes the *-B* and *-L* options as *"Force output to be written with Big-Endian byte order"* and *"Force output to be written with Little-Endian byte order."* Although not explicitly declared, the conflict between *-B* and *-L* can be inferred by comparing the two sentences, which depict two diametrically opposite behaviors. An accurate and straightforward way to identify these relationships is manual inspection, but it is labor-intensive and unrealistic for large-scale identification. How to find out relationships among options from documents automatically becomes necessary.

Challenge - C2. After relationships in natural language form have been identified from documentation, it is still hard to automatically extract concrete relationships (e.g., conflicts or dependencies). First, it is challenging to locate a relationship's related options automatically. For example, the sentence, *"Either -f or -b must be used with -C, and -C cannot be used with -F or -d,"* declares a dependency among the *-f*, *-b*, and *-C* options, along with a conflict among the *-C*, *-F*, and *-d* options. These relationships' related options cannot be automatically determined without an accurate analysis of the grammatical structure. Second, as mentioned before, the relationship among options can be declared in entirely different ways. So simple methods like keyword matching cannot be used to determine the kind of relationship declared automatically.

In this paper, we addressed the above challenges and proposed CarpetFuzz, an NLP-based fuzzing assistance tool for extracting program option constraints. The basic idea of CarpetFuzz is to use natural language processing (NLP) to identify and extract the relationships (e.g., conflicts or dependencies) among program options from the description of each option in the documentation and filter out invalid combinations to reduce the option combinations that need to be fuzzed. Given a program's document, CarpetFuzz first extracts all its options and corresponding descriptions by parsing the OPTIONS section. Then CarpetFuzz uses a machine learning model to determine whether a relationship is declared in the descriptions. Since such sentences account for very little in the document (e.g., 3.4% in the tiffcp document), we use the entropy-based uncertainty sampling [11], an effective active learning methodology, to reduce human effort in labeling training data. To identify relationships implicitly declared by multiple sentences, CarpetFuzz summarizes a series of features of implicit declarations and leverages NLP to find all sentence pairs satisfying these features. After identifying relationships among options in the program, CarpetFuzz leverages a mixing forward and backward traversal method to find the relationship-related node from the dependency tree (obtained from the dependency parsing). Furthermore, based on linguistics, CarpetFuzz leverages a polarity-based finite state machine to determine concrete relationship. At last, CarpetFuzz filters out combinations not satisfying these relationships to reduce the number of combinations for fuzzing.

We evaluated CarpetFuzz on 20 popular real-world open-source programs. According to their documents, CarpetFuzz extracted 282 relationships from the documents which include 2952 sentences in 260.8 seconds with 96.10% precision and 88.85% recall. Among these relationships, 218 were implicitly declared, with 95.87% precision and 90.09% recall. Based on these relationships, we reduced the 67.91% option combinations, significantly reducing the search space of combinations, which demonstrated the necessity of extracting option relationships. With the valid option combinations, CarpetFuzz helped AFL find 45.97% more paths that other fuzzers cannot discover. Also, CarpetFuzz found 57 crashes. After our analysis, 30 crashes were related to specific option combinations, proving the importance of fuzzing with valid option combinations. So far, 30 crashes have been assigned with CVE IDs. We also compared the fuzzing performance of CarpetFuzz with the state-of-the-art option configuration fuzzing tool POWER on its benchmark. CarpetFuzz found 94 unique crashes, 1.71 times that of POWER.

Responsible disclosure. We immediately disclosed all these vulnerabilities to the software developers as soon as we discovered them. Moreover, we have proactively provided patches to them. Till now, 45 vulnerabilities have been fixed, nine of which were fixed by our patches.

Contributions. The contributions of this paper are summarized as follows:

• *New technique.* We proposed a novel technique for identifying and extracting constraints among program options from the documentation. To the best of our knowledge, this is the first study that tries to use NLP to automatically figure out the relationships among program options from the documentation. With the help of this technique, AFL finds 45.97% more paths that other fuzzers cannot discover.

• *Implementation and discoveries.* We implemented the prototype tool, CarpetFuzz, and evaluated it on 20 popular real-world open-source programs. CarpetFuzz accurately extracted 88.85% of the relationships from their documents. Through fuzzing these programs with the valid option combinations obtained by CarpetFuzz, 57 unique crashes have been found, 30 of which have been assigned with CVE IDs.

• *Releasing code and data.* We open-sourced our prototype tool and dataset to promote research in this domain[1].

[1]Available in `https://github.com/waugustus/CarpetFuzz`

2 Background and Related Work

2.1 Coverage-Guided Fuzzing

The coverage-guided fuzzing is a technique that leverages the code coverage information obtained from the program instrumentation to guide the fuzzing process. Like the traditional black-box fuzzing technique, the coverage-guided fuzzing technique tries to trigger an exception in the target program by keeping mutating the input, but it has a more effective mutation process due to the coverage information. The process of coverage-guided fuzzing is shown in Figure 1. To start a coverage-guided fuzzing, users need to specify the options (in the command line) and the seed file (step 1). The options are stored in the configuration as a constant and will not change in the whole fuzzing process. The seed file is passed into the input queue and then sent into the mutation engine to change its content (step 2). After mutation, the execution engine reads the execution command from the configuration and replaces the input with the mutated file to check its coverage (step 3). If the mutated file activates new coverage (i.e., interesting), this mutated file will be added to the input queue for further mutation (step 4).

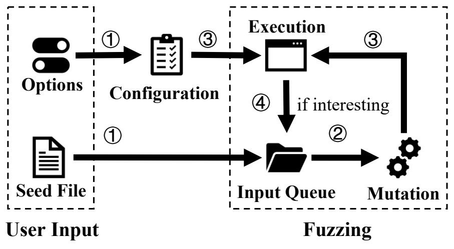


Figure 1: Process of Coverage-guided Fuzzing.

Many studies have focused on improving coverage-based fuzzing by modifying the seed selection [20,41], seed schedule [3,4,8,32,40,54,61,67], and feedback strategies [1,6,15,16,30,33,44]. Among them, Rebert et al. [41] suggested six seed selection approaches and formalized the problem as set cover and integer linear programming. Böhme et al. [4] formalized the coverage-guided fuzzing process with a Markov Chain model and sped it up by increasing the number of mutations of seeds with low-frequency paths. Chen et al. [6] proposed a fine-grained feedback metric leveraging byte-level taint tracking to determine the branch-related byte. These works significantly improved the fuzzing efficiency and increased the coverage in the fuzzing process. However, in our tests, these fuzzers cannot reach the option-related paths.

2.2 Option and Combination

A command-line option is an item of information that the user specifies when the program starts to tell it which operation to enable or disable [58]. A program usually has multiple options, which follow the program name in the command line and are separated by spaces. For example, *tiffcp* and *pdftops* have 20 and 34 options, respectively. With the help of these options, users can pass their demands to the program so that it can do what they want.

When the demands become complex, users need to specify multiple options simultaneously instead of using a few options. However, not all combinations are valid due to the relationships (e.g., conflicts and dependencies) among options. An invalid combination may make the program throw an exception and exit at an early stage. To prevent users from using invalid combinations, developers usually note the relationships among options in a natural language form in the documentation, such as *"make sense only when -p option is specified,"* and *"The -level1 option cannot be used with -form."*

To this end, some studies have sought unreachable paths by combining options (or called option configurations). AFLargv [5] mutates option configurations within the lower and upper bounds of the number and size of command-line options. Wang et al. [55] applied option-aware fuzzing to directed grey-box fuzzing to reach unreachable target locations through option configuration search. Song et al. [45] proposed a new coverage feedback metric, validity pair, to predict whether the parser rejects an input to check the validity of execution. Zeller et al. [64] designed a tool to automatically infer option configuration from source code based on specific option parsing modules (e.g., the *getopt* function in C). POWER [29] utilized three mutation operations to explore option configurations. However, due to the lack of considering the constraint among options, many combinations generated by most works may be invalid, reducing the search efficiency. ConfigFuzz [66] considered such constraints but relied on manual inspection. Although extracting constraints among options may be a one-time job, it requires testers to understand target programs in-depth, which is unaffordable, especially when performing large-scale testing (e.g., thousands of applications). Inspired by this finding, we studied automatic constraint extraction from the documentation to filter out invalid option combinations.

2.3 NLP-based vulnerability discovery.

There are also previous works trying to discover vulnerabilities based on NLP, which mainly found programs' constraints from the code comments [17,48–50,59,65] and documentation [31,38,68] to assist with static analysis. Furthermore, Xie et al. [60] tried to extract information from the documentation to guide fuzzing the deep learning (DL) libraries. However, this work only applied to API libraries, not executable programs. Moreover, it relied on strictly formatted documents (i.e., DL documents) and cannot be used to parse loosely formatted documents such as manpages. Different from previous works, we tried to leverage NLP to parse loosely formatted documents and extract constraints from the documents to filter out invalid option combinations to assist with fuzzing.

3 Design

In this section, we describe the design of CarpetFuzz, an NLP-based fuzzing assistance tool for extracting program option constraints. The core idea is to use NLP to identify and extract the relationships (e.g., conflicts or dependencies) among program options from the description of each option in the documentation and filter out invalid combinations to reduce the option combinations that need to be fuzzed. We first introduce a general overview of the whole design and then show how each component works.

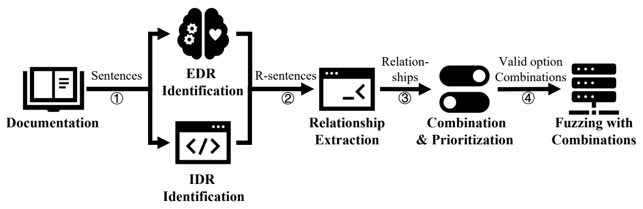


Figure 2: Overview of CarpetFuzz. EDR: explicitly declared relationships, IDR: implicitly declared relationships.

3.1 Overview

Figure 2 shows the overview of CarpetFuzz. Given a document of the target program, CarpetFuzz first extracts all its options and corresponding descriptions by parsing the OPTIONS section, and splits these descriptions into sentences with NLP tools (step 1). Then CarpetFuzz identifies sentences containing relationships (called R-sentences [2]) among these options (step 2). Specifically, with active learning algorithm, CarpetFuzz builds a dataset from Linux manpages and trains a machine learning model to identify explicitly declared relationships. To find the implicitly declared relationships, CarpetFuzz combines the NLP parsing technique (dependency parsing and constituency parsing) with our heuristic rules. After identifying R-sentences, CarpetFuzz builds the dependency trees of these R-sentences with the dependency parsing and traverses the dependency tree from the node where the option is located to extract the concrete relationships and objects (step 3). With the extracted relationships, CarpetFuzz builds valid option combinations and prioritize them by coverage (step 4). At last, CarpetFuzz passes these prioritized valid option combinations to the fuzzer.

[2]In the rest of this paper, we use R-sentences to refer to sentences containing relationships

3.2 Explicitly Declared Relationship Identification

After analyzing a large number of documents, we find that the explicitly declared relationships between options can mainly be divided into five categories, including conflict, dependency, implication, similarity, and supersedence (Appendix A). The conflict among options represents that these options must not be used together, and the dependency among options represents that these options must be used together. Implication indicates that the function of one option includes another option. Similarity and supersedence indicate that the options' functions are, respectively, roughly the same and replaceable. All these three relationships indicate an overlap exists in multiple options' functions. Although the combination of options with these three relationships may not cause an exception, it makes some options overwritten. Fuzzing with the combinations of these options may have the same effect as fuzzing with each and may be useless for discovering new paths. To avoid such useless combinations and reduce the search space, we treat these three relationships as conflicts in this paper.

We collect extensive documentation from the internet and extract the sentences in all options' descriptions as the unlabeled dataset. To reduce the labeling cost, we use an active learning algorithm to manually label part of the samples in the unlabeled dataset and add them to the labeled dataset. Specifically, we first read a small set of the documents and manually collected multiple keywords (about 20) in R-sentences (e.g., "combine with," "imply," "like," "ignore"), which was a small one-time job (about 5 minutes). Note that although these keywords were from a few documents, they also applied to other documents and could be systematically augmented through data labeling in active learning. Then we sample a subset of the sentences containing these keywords for manual labeling, which is the initial training dataset of our model. Sentences with the above five relationships are labeled as positive, and other sentences are labeled as negative. Generally, hundreds of labeled data are enough to train the initial model, independent of the dataset's size. We utilize the word2vec model [42] to map words into vectors as the input features because of its simplicity and low-cost. We have also evaluated more advanced models (e.g., BERT [13]) and obtained a similar performance. In each iteration of the active learning algorithm, the prediction of all samples in the unlabeled dataset is obtained using the machine learning model, and the samples are selected for manual labeling according to the entropy-based uncertainty sampling algorithm [11],

$$e_i = -\sum_{j=0}^{1} P(y_j|x_i) \cdot \log P(y_j|x_i) \quad (1)$$

where e_i is the entropy of the i-th unlabeled sample, $P(y_1|x_i)$ represents the predicted probability that x_i belongs to class 1 (is an R-sentence), and $P(y_0|x_i)$ represents the predicted probability that x_i belongs to class 0 (is not an R-sentence). Higher entropy represents higher uncertainty when the model predicts the sample. Thus labeling those samples helps update the model more effectively. *K* samples with the highest entropies are manually labeled and added to the training set, and the model is retrained using the updated training set (Appendix B). After *T* iterations, the final model will be used for explicitly declared relationship identification. In particular, we manually labeled 1,381 sentences (557 positive and 824 negative sentences) in the active learning process, which only account for 0.46% of all the unlabeled sentences. Note that data labeling is a one-time effort (only performed in the training process, about 5 hours), and later testing does not require human effort. We randomly sampled 1,000 data from the remaining unlabeled dataset for evaluation, and the accuracy, false-positive rate, and recall of the final model were 92.90%, 11.49%, and 98.42%, respectively.

3.3 Implicitly Declared Relationship Identification

As mentioned in the introduction, it is challenging to identify implicitly declared relationships. These relationships involve multiple options whose behaviors are different but related and can only be identified when the association is found. After analyzing the documentation of a large number of programs, we have only found implicitly declared conflicts (implicitly declared dependency has not been found yet). In this paper, we only discuss the identification of implicitly declared conflicts. We find that the sentences implicitly declaring a conflict (i.e. implicit R-sentences) among different options usually have the same objects and same/antonyms verbs, implying that these options do the same (or opposite) things on the same object. Their grammatical structures usually satisfy parallel structures (i.e., repetition of the same grammatical form in several parts), which may be because the document's authors copied the description for each conflict option for convenience. For example, the descriptions of the *-B* and *-L* options are *"Force output to be written with Big-Endian byte order,"* and *"Force output to be written with Little-Endian byte order."* Since these two sentences have the same predicate (i.e., force), object (i.e., output), and grammatical structure (parse trees), we can determine that they are implicit R-sentences.

To determine whether several sentences are implicit R-sentences, we need to extract the objects, predicates, and the parse trees of their descriptions. Figure 3 shows the process of extracting these features. We first preprocess the descriptions (step 1) by extracting the first sentence of each description for analysis because such a sentence is usually the topic sentence introducing what the option does. We find that some topic sentences for options may have no subject (begins with a verb), causing mistakes in the NLP parser, such as misjudging the verb "force" as a noun. Therefore, for a sentence that begins with a verb, we restore its subject (i.e., it) and modify the verb's person to avoid parsing errors. For example, the description *"Force output to be written with Big-Endian byte order"* of the *-B* option in *tiffcp* will be modified as *"It forces output to be written with Big-Endian byte order."*

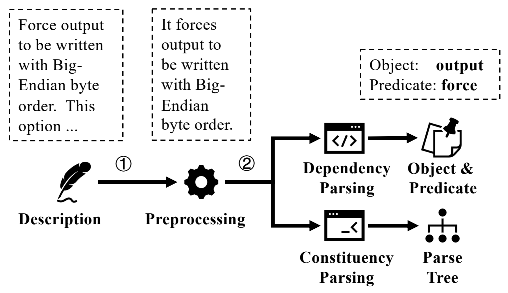
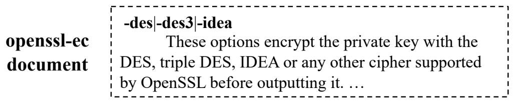

Force output to be written with Big-Endian byte order. This option ...
It forces output to be written with Big-Endian byte order.
Object: output
Predicate: force
①
②
Description
Preprocessing
Dependency Parsing
Object & Predicate
Constituency Parsing
Parse Tree

Figure 3: Process of extracting the features for identifying implicit R-sentences.

After the preprocessing, sentences will be analyzed with dependency parsing to find predicates and direct objects. (step 2). Specifically, we use NLP tools to label each word's dependency tag and find the words with the tag *"dobj"* (i.e., the direct object) and *"ROOT"* (i.e., the predicate). To determine if these two sentences have parallel structures, we use constituency parsing to build their parse trees and remove their leaf nodes since leaf nodes represent the specific words instead of grammar structures. If their lengths are the same, we judge them as parallel when their remaining trees are identical. If their lengths are different, we first traverse the shorter tree from its last node to find its nearest branch node and then remove this branch node and all of its children nodes. If the remaining tree is a subtree of the longer tree, we regard these two trees are in parallel structures. Finally, several sentences are considered as implicit R-sentences when they have the same object, the same or mutually antonymous predicates, and parse trees with parallel structures.

Moreover, in some documents, multiple options are written together in the same position, which is another kind of implicitly declared conflict. For example, the *-des*, *-des3*, and *-idea* options in *openssl-ec* are written together as *"-des|-des3|-idea,"* as shown in Figure 4. However, options written together are not necessarily conflicting - they may be aliases for the same option. For example, although the *-s* and *–silent* options are written together as *"-s, –silent,"* they are not conflicting but the same option. We can determine whether a topic sentence shared by multiple options is an implicit R-sentence by checking its subject. When the subject is plural (e.g., "they" or "these options"), it means that this sentence is an implicit R-sentence.



openssl-ec document

-des|-des3|-idea
These options encrypt the private key with the DES, triple DES, IDEA or any other cipher supported by OpenSSL before outputting it. …

Figure 4: Example of options written in the same position.

Altogether, we discovered 218 implicitly declared conflict pairs from 20 popular program documents. The precision and recall were 95.87% and 90.09%, respectively.

3.4 Relationship Extraction

After finding out the R-sentences from the documentation, we need to extract the specific relationships from these sentences, including conflicts and dependencies (As mentioned in Section 3.2, implication, similarity, and supersedence are also treated as conflicts).

Extracting explicitly declared relationships. For R-sentences of explicitly declared relationships, we first divide complex sentences into multiple clauses with constituency parsing. If a clause contains options, we replace their names with custom symbols (e.g., *option_A*) to avoid the minus sign interfering with parsing. We then use dependency parsing to obtain the parse tree of this clause. Figure 5 shows the extraction process of explicitly declared relationships based on the parse tree. First, we locate where the first option occurs and traverse the parse tree forward to see if any other option has co-dependencies with the first option (i.e., the *"conj"* tag). Since R-sentences may contain other programs' options, we check if the located option belongs to the target program before analyzing. We treat options with co-dependencies as a single option and confirm the conjunctions (i.e., the *"cc"* tag). For options without co-dependencies, we perform the relationship extraction process for them separately. Second, we traverse the parse tree from the option's position backward to find the verb or adjective related to the option (i.e., the *"pobj"* and *"prep"* tags). We then compare this word to our keywords, which are collected from many documents and augmented based on synonyms, to confirm if it is a keyword. If not, we search for all its synonyms and compare them to the keywords. Third, we traverse the parse tree from the verb's position backward to find the negations and modals (i.e., the *"aux"* tag) related to this verb for determining the concrete relationships.

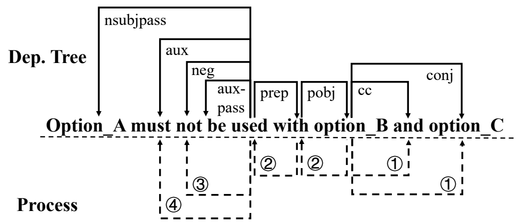


Figure 5: Example of dependency parsing and extraction process of explicitly declared relationships. The part above the dotted line is the parse tree from dependency parsing, the text on the bold line represents the dependency tag, and the arrow represents the dependency relationship. The part below the dotted line shows the relationship extraction process, where the arrow indicates the traversal direction.

As mentioned above, relationships can be declared in entirely different ways. For one thing, the same relationship can be declared by various keywords. For another thing, the relationship described by the same keyword may vary in different contexts. For example, the keyword "use" describes the dependency, neutrality (no relationship), and conflict, respectively, in *"must be used," "be used,"* and *"must not be used."*

Based on linguistics [25], conflict and dependency can be regarded as a pair of polar opposite items, and neutrality is the intermediate (i.e., neither conflict nor dependency) between the two polarities. These three items can be transitioned according to specific sentence components, like keywords, deontic modals (i.e., modals indicating obligation and permission, like *"must," "should,"* and *"have to"*), and negations. We use two finite state machines (FSMs) to describe the state transition process, as shown in Figure 6. The choice of FSM is related to the semantic-based classification of keywords, including conflicting and dependency keywords (Appendix C). Each FSM contains three states, S_D, S_N, and S_C (i.e., dependency, neutrality, and conflict), and two possible initial states. The initial state of FSM is the default relationship of the keyword (Appendix C), which is expressed without deontic modals and negations. For example, since *"be used"* expressed neutrality, the default relationship of the keyword *"use"* is neutrality. After determining the initial state, we can infer the final state based on the deontic modals and negations in the sentence, which is the relationship expressed. Note that, based on linguistics [22], deontic modals take scope under negations, which means that only negations have an effect when both are present [3]. We take the sentence, *"-A must not be used with -B and -C,"* as an example. Since the keyword *"use"* is a dependency keyword, the bottom FSM is chosen, and the initial state is neutrality. Since the sentence has both deontic modal and negation, only the negation will have an effect. From the FSM for dependency keywords, the state will be transitioned from neutrality to conflict.

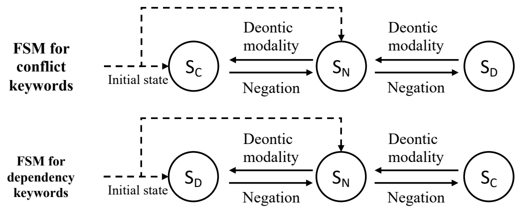


Figure 6: FSMs for conflict and dependency keywords. The dotted lines point to possible initial states.

Extracting implicitly declared relationships. For implicit R-sentences, they are determined to be declarations of conflicts, and we only need to map these sentences to the corresponding options, which is already done in step 1 in Figure 2. For example, based on the method in Section 3.3, we can know that the

[3] An even number of negations will cancel each out.

sentences *"Force output to be written with Big-Endian byte order"* and *"Force output to be written with Little-Endian byte order"* are implicit R-sentences. According to the correspondence between the options and the sentences, we can know that these two sentences belong to the *-B* and *-L* options, respectively. Finally, we extract the conflict relationship between the *-B* and *-L* options.

Finally, with our relationship extraction method, we successfully extracted 282 relationships from 20 popular program documents with a precision and recall of 96.10% and 88.85%.

3.5 Combination and Prioritization

Based on the extracted relationships between options, we could filter out all invalid option combinations. Specifically, we combine all options with length n ($0 \leq n \leq k$, where k is the number of options) and perform a validity check on the generated combinations against the extracted relationships. A combination is considered valid only if it satisfies all dependencies and has no conflicts. For options that can have values, we manually collected reasonable values for each option from the documentation and randomly picked one value for each option in each combination.

Although the invalid combinations are filtered, which make up most of the total combinations (e.g., 99.84% in *openssl-rsa*), the number of the left valid combinations may still be large (e.g., millions), and it is impractical to test all these combinations, which would require enormous computing resources. In this paper, we leveraged N-wise testing [57] to further prune the combinations that need to be tested. This testing is an effective method for combinatorial interaction testing which greatly reduces the combinations that need to be tested by focusing on defects caused by the interaction of N factors. According to the research, almost all flaws are caused by the interaction of no more than six factors [27]. So we used 6-wise ($N = 6$) testing to prune the valid option combinations.

After pruning the combinations, we prioritize each combination based on its coverage in the dry run on the same seed file. Specifically, combinations with higher coverage will have a higher priority. We use this prioritization technique for two primary considerations. First, fuzzing with combinations with higher coverage in the dry run is more likely to discover new paths during the fuzzing process. Second, for some invalid combinations not identified in previous steps, our prioritization technique can lower their priority, reducing their chances of being tested.

Fuzzing with combinations. Finally, we fuzzed each program with all the prioritized pruned option combinations. Specifically, we instrument the target program to allow it to read options from a file and let the fuzzer modify the file on the fly to switch the combinations in use. At the beginning of fuzzing, we use all the given combinations to mutate the seed files and record the corresponding combination when generating a new test case. Then we use the corresponding combination to mutate each test case in the queue.

4 Implementation

This section introduces the implementation of the prototype of CarpetFuzz in our research, including dataset collection, model training, NLP analysis, prioritization, and fuzzing.

Dataset collection. We collected the training dataset by crawling the manpages of all command-line programs (37,672) from the Debian Manpages Project [12], the complete repository of all manpages contained in Debian. Note that documentation exists in three forms, manpages, online documentation, and help command. Considering that online documentation is usually generated from manpages, and the output of help commands is generally brief and does not have the information we need due to space limitations, we selected the manpages to parse. After crawling these manpages, we extracted the content of the options sections based on the GNU roff language (i.e., Groff) [56] and mapped each option to its description. Specifically, we removed all format tags defined in Groff and distinguished different options by paragraph separators (the *".TP"* and *".IP"* tag). We treat the content between multiple options as their descriptions. Then we used spaCy [47] for sentence segmentation. Finally, we collected 302,875 sentences related to options (228,827 after deduplication).

Model training. We utilized the Word2Vec model in the Gensim library [42] to map words into vectors, trained with the following parameters: $size = 300$, $window = 5$, and other default settings. Since the description of an option may contain itself, we convert the option names in the sentence to *option_itself* and *option_other* during preprocessing. We used XGBoost to classify sentences related to options as we experimentally found that XGBoost [7] had a better performance than other machine learning models (e.g., SVM [10] and RF [21]). We utilized the default hyper-parameters for training the initial model on the initial dataset (297 positive and 139 negative sentences). At every iteration of the active learning process, we selected 20 ($K = 20$) sentences with the highest entropy for manual labeling and retrained our model with the optimal hyper-parameters from the grid search [28]. We stopped training after 70 consecutive iterations (i.e., $T = 70$). The hyper-parameters of the final model were: $max\_depth = 6$, $n\_estimators = 200$, $colsample\_bytree = 0.8$, $subsample = 0.8$, $learning\_rate = 0.1$, and other default settings.

NLP analysis. We used NLTK [39] to output all parts of speech for a word and LemmInflect [2] to convert a verb to third-person singular after adding the subject. We utilized spaCy for sentence segmentation to extract the first sentence and dependency parsing to label each word's dependency tag. For constituency parsing, we used the AllenNLP library [24] to build the parse trees and extracted clauses based on the *"SBAR"* tag. In the analysis, we found that different numbers lead to different dependency parsing results, which is a flaw of

the NLP model. As a fix, we removed all non-option numbers.
Prioritization and Fuzzing. We used the PICT tool [34] to implement the 6-wise testing and automatically generate model files based on the extracted relationships to specify limitations on combinations. We used afl-showmap [62] to count edge coverage information for a dry-run as a sorting criterion and LLVM pass [53] to instrument the target program.

5 Evaluation

This section describes our evaluation of CarpetFuzz, including the effectiveness of both its end-to-end operation and individual components.
Real World Dataset. In this evaluation, we evaluated the effectiveness of CarpetFuzz on the latest versions of 20 popular real-world programs (Appendix D). These programs handle 11 different types of input files, including image (*TIFF* and *JPG*), certificate (*PEM*), text (*MD* and *JSON*), traffic packet (*PCAP*), executable file (*ELF*), archive (*LRZ*), audio (*OGG* and *SPX*), and document (*PDF*). We chose these programs because they were widely used programs with more than ten options and were continuously maintained. We collected their latest manpages from the compilation directory (i.e., "*share/man/man1/*") and manually extracted all R-sentences and relationships for evaluation. The number of options and relationships [4] for each program [5] are shown in Appendix D.
Experimental Setup. We used CarpetFuzz to augment AFL [63] and compared it to the original AFL to demonstrate how CarpetFuzz could help with it. In order to highlight the effect of CarpetFuzz, we also selected the latest versions of AFLfast [4], MOPT-AFL [32], and AFLplusplus [14] for comparison, which were among the most popular improved fuzzers based on AFL. Each fuzzer started with the same single seed file collected from AFL directory, test directory (for *PEM* and MD format), or online corpus [35] (for *OGG* and *SPX* formats). We continuously fuzzed our test programs for 48 CPU hours and repeated the experiment five times to avoid the effects of the inherent uncertainty of fuzzers mentioned in [26]. Our experiments were performed on a machine of Intel Xeon Platinum 8268 with 24 CPU cores and 188GB RAM, which runs Ubuntu 20.04.5 LTS.
Research Questions. In the following sections, we aim to answer the following research questions:
RQ1. What is the performance of CarpetFuzz?
RQ2. What is the accuracy of relationship identification?
RQ3. What is the accuracy of relationship extraction?
RQ4. What is the effectiveness of CarpetFuzz's prioritization technique?
RQ5. What is the fuzzing performance of CarpetFuzz compared to the state-of-the-art techniques?
RQ6. Can CarpetFuzz discover real-world vulnerabilities?

5.1 Performance of CarpetFuzz (RQ1)

In this experiment, we aimed to evaluate the performance of CarpetFuzz. To highlight the improvement of the performance of AFL by CarpetFuzz, we defined the evaluation metrics as the number of edges only covered by CarpetFuzz.

From the 20 programs' manpages, CarpetFuzz extracted 282 relationships and filtered out 67.91% of the option combinations on average (Appendix E). With the relationships, CarpetFuzz reduced the option combination to be tested in the 20 programs by 25.00% to 99.85%. It can be seen that for programs with complex relationships among options (e.g., *openssl-rsa* and *eu-elfclassify*), CarpetFuzz can greatly reduce the number of combinations to be tested. We manually labeled the relationships in these programs to evaluate the process of filtering out invalid option combinations. The precision and recall ranged from 68.46% to 100% and from 70.00% to 100%, respectively, and the average values were 98.01% and 94.19%, demonstrating that CarpetFuzz could accurately identify and filter out invalid option combinations. Note that only a few misjudgments occurred in *pdftotext* and *tiffcp* (Section 5.3) but accounted for a large proportion of all relationships extracted due to a small number of total relationships, which led to a low precision/recall (68.48%/70.00%).

After filtering the invalid option combinations, we used the 6-wise testing mentioned in Section 3.5 to prune the remaining combinations further. To evaluate whether pruning would cause a loss of coverage, we randomly sampled 100,000 valid combinations for each program for comparison (Appendix F). The results showed that our pruning technique could reduce much more combinations (98.91%) compared to random sampling while only slightly losing edges (2.54%).

We then used AFL, AFLfast, MOPT-AFL, and AFL++ to fuzz each program without options (or with the minimum options to make it work) and used CarpetFuzz to fuzz with the pruned option combinations in order of priority. The growth curves of the code coverage of each fuzzer converged within 48 hours , representing that the coverage of these fuzzers was eventually stabilized, and using 48 hours of coverage basically represents the coverage capability of these fuzzers per run.

We used the edges that AFL could reach as the baseline to evaluate CarpetFuzz's performance. Due to the inherent uncertainty, the coverage of each run of the same fuzzer may be very different, so the single coverage cannot represent the coverage capability of the fuzzer. Instead, we took the union of the coverage of the five runs to represent the edges that the fuzzer is capable of reaching. We defined the number of unique edges (edges unreachable for the baseline) of the test fuzzer n and the ratio of unique edges r as

$$n = |E_f - E_f \bigcap E_b| \qquad r = \frac{|E_f - E_f \bigcap E_b|}{|E_b|} \tag{2}$$

[4] In order to facilitate comparison, we split the relationship among multiple options into multiple relationships between every two options.

[5] We have removed all help, version options, and options requiring specific support that we cannot satisfy (e.g., exceptional file support and OPI support).

where E_f represents the union of the fuzzer's edge coverage in five runs, and E_b represents the union of the baseline's edge coverage in five runs. Then we counted the number and ratio of unique edges for each fuzzer, as shown in the first 14 columns of Table 1. With the AFL as the baseline, AFLfast found the fewest unique edges, and the ratio of unique edges on these 20 programs was 1.31% on average, which may be because AFLfast mainly improved AFL's seed schedule strategy to speed up the convergence of the coverage curve. MOPT and AFL++ found much more unique edges than AFLfast, and the ratios of unique edges of these two fuzzers were 4.47% and 5.39%, respectively. We thought that MOPT and AFL++ performed much better than AFLfast, probably because they improved the mutation strategies of AFL. However, even without optimizing any strategy of AFL, CarpetFuzz found the most unique edges, and the average ratio of unique edges was 47.23%. On all these 20 programs, the ratio of CarpetFuzz was higher than other fuzzers and was 336.20% at most (*lrzip*), which was at least 293.33 times (AFL++) the ratio of other fuzzers. The results showed that for some programs, such as *eu-elfclassify* and *jpegotim*, few unique edges might be discovered by improving the fuzzing strategy, but this can be alleviated by specifying certain valid option combinations.

We also took the union of the edges of the first four fuzzers to investigate how many edges of CarpetFuzzcould not be discovered by other fuzzers, as shown in the last three columns of Table 1. The results showed that an average of 94.59% of the unique edges of CarpetFuzz were not discovered by other fuzzers, and CarpetFuzz could help AFL find 45.97% more edges that other fuzzers cannot discover on average.

5.2 Accuracy of Relationship Identification (RQ2)

Accuracy of identifying explicitly declared relationships. During the active learning process, we performed 70 iterations and collected 1,817 manually labeled data as the training set. We used the final model to predict all sentences in the dataset (except the training and validation sets), with 7,483 sentences predicted as positive and 221,344 sentences predicted as negative. We randomly sampled 500 sentences from each category and manually labeled them for evaluation. The model's accuracy on the evaluation dataset was 92.90%, indicating that it has high accuracy. The false-positive rate and recall were 11.49% and 98.42%, implying that our method had a lower risk of identifying R-sentences as non-R-sentences and, thus, may avoid missing some relationships.

We then evaluated our model on the sentences of the 20 programs' manpages. According to the results of manual labeling, 75 sentences were explicit R-sentences among the 2,952 sentences from the documents. Our model predicted 76 sentences to be positive, including 9 false positives and 8 false negatives. The accuracy is 98.80%, and the false-positive rate and recall were 0.67% and 89.33%, respectively. Although the accuracy of this dataset was higher, its F1 score was lower (0.89) than the former evaluation dataset (0.92). The main reason may be that this dataset was imbalanced. Note that eight of the nine false positives were judged as neutrality and ignored in the relationship extraction process (described in Section 3.4), which did not affect the extraction results.

Accuracy of identifying implicitly declared conflicts. From these documents, we found 232 unique conflict pairs declared implicitly. We performed an identification with the method in Section 3.2 and found 218 such conflict pairs, among which the false positives and false negatives were 9 and 23, respectively. The precision was 95.87%, and the recall was 90.09%, implying that our method had a good performance in identifying implicitly declared conflicts. After our analysis, these nine false positives were all due to a lack of relevant common sense. For example, the topic sentences of *-X* and *-Y* options in *tiffcrop* were *"Set the horizontal/vertical dimension of a region to extract relative to the specified origin reference."* Although both options were to *"set the dimension,"* our tool did not know that horizontal and vertical dimensions were different, requiring human common sense. Seven false negatives were also due to a lack of relevant common sense. In the topic sentence of the *-s* option in *jpegoptim*, *"Strip all markers from output file,"* our tool did not know that the *"all marker"* referred to all kinds of markers rather than a specific marker. So it missed the implicitly declared conflicts between the -*s* option and options to strip single kinds of markers. Seven false negatives came from the inaccurate parsing results of the NLP parser underlying our implementation (e.g., SpaCy). For example, in two topic sentences with the same structure, *"output monochrome sixel image"* and *"output 15bpp sixel image,"* the parser judged the word *"monochrome"* as a noun while regarding *"15bpp"* as a number, leading these sentences to be mistaken as non-parallel and thus identified as non-conflicting. Nine false negatives were due to slight differences in topic sentences. For example, the topic sentences of *-pvk-strong* and *-pvk-none* options were *"Enable Strong PVK encoding level"* and *"Don't enforce PVK encoding."* Since both the verbs (enable versus enforce) and objects (level versus encoding) differed, our tool cannot identify the implicitly declared conflict from these sentences. Interestingly, we found that such missing conflicts were identifiable from their names.

5.3 Accuracy of Relationship Extraction (RQ3)

After manual labeling, we found 305 relationships from the 20 programs, including 284 conflicts and 22 dependencies. Our tool found 282 relationships, of which 264 were conflicts and 18 were dependencies. The precision and recall of conflict were 95.83% and 89.40%, and those of dependency were 100% and 81.82%. The precision and recall of all relationships were 96.10% and 88.85%, implying that our tool performed well in extracting relationships. We found that dependencies

Table 1: Results of each fuzzer with 48 hours. E_b: the union of the baseline's edge coverage in five runs, E_f: the union of the fuzzer's edge coverage in five runs, n: the number of unique edges, r: the ratio of unique edges.

Program	Baseline										CarpetFuzz					
	AFL	AFLfast			MOPT			AFL++			$\|E_f\|$	Compare to AFL		Compare to ALL		
	$\|E_b\|$	$\|E_f\|$	n	r	$\|E_f\|$	n	r	$\|E_f\|$	n	r		n	r	n_{all}	r_{all}	**Pct.**
cmark	6308	6229	106	1.68%	6608	372	5.90%	6860	552	8.75%	7080	960	**15.22%**	876	13.89%	91.25%
editcap	3340	3201	140	4.19%	3749	465	13.92%	3755	464	13.89%	3800	635	**19.01%**	550	16.47%	86.61%
eu-elf classify	140	134	0	0.00%	140	0	0.00%	140	0	0.00%	194	54	**38.57%**	54	38.57%	100.0%
img2sixel	1584	1635	84	5.30%	1981	440	27.78%	2055	494	31.19%	2172	710	**44.82%**	685	43.24%	96.48%
jpegoptim	165	158	0	0.00%	165	0	0.00%	165	0	0.00%	284	120	**72.73%**	120	72.73%	100.0%
jpegtran	4695	4708	49	1.04%	4845	155	3.30%	4995	301	6.41%	4339	898	**19.13%**	875	18.64%	97.44%
jq	1910	1900	14	0.73%	1895	9	0.47%	1935	25	1.31%	2255	374	**19.58%**	365	19.11%	97.59%
lrzip	1047	1029	2	0.19%	1051	4	0.38%	1057	12	1.15%	4559	3520	**336.20%**	3518	336.01%	99.94%
ogg123	321	321	0	0.00%	320	0	0.00%	325	5	1.56%	363	42	**13.08%**	41	12.77%	97.62%
openssl-asn1parse	2109	2109	0	0.00%	2111	2	0.09%	2111	2	0.09%	2599	490	**23.23%**	489	23.19%	99.80%
openssl-ec	5015	5036	37	0.74%	5029	29	0.58%	5058	57	1.14%	7578	2601	**51.86%**	2561	51.07%	98.46%
openssl-rsa	5350	5337	27	0.50%	4994	33	0.62%	5441	95	1.78%	6291	1135	**21.21%**	1106	20.67%	97.44%
pdftops	984	999	17	1.73%	1002	19	1.93%	1005	21	2.13%	1059	76	**7.72%**	68	6.91%	89.47%
pdftotext	969	975	8	0.83%	974	5	0.52%	977	8	0.83%	1026	62	**6.40%**	60	6.19%	96.77%
podofo encrypt	236	239	3	1.27%	239	3	1.27%	239	3	1.27%	248	12	**5.08%**	12	5.08%	100.0%
speexdec	495	495	0	0.00%	548	53	10.71%	497	2	0.40%	594	99	**20.00%**	99	20.00%	100.0%
tcpprep	442	435	0	0.00%	454	12	2.71%	441	0	0.00%	883	454	**102.71%**	453	102.49%	99.78%
tcpreplay	369	364	3	0.81%	373	9	2.44%	400	39	10.57%	657	315	**85.37%**	298	80.76%	94.60%
tiffcp	4570	4348	140	3.06%	4930	474	10.37%	5057	721	15.78%	5544	1048	**22.93%**	724	15.84%	69.08%
tiffcrop	5152	5049	211	4.10%	5309	327	6.35%	5522	489	9.49%	5814	1018	**19.76%**	809	15.70%	79.47%
Average				1.31%			4.47%			5.39%			**47.23%**		**45.97%**	**94.59%**

may be more challenging to extract, resulting in a lower recall and higher precision. Among the four false negatives of dependencies, two were due to the false negatives of the model, and two were due to the inability to find the corresponding options. For example, our model identified the R-sentence in *pdftotext*, *"This is ignored in all other modes,"* but our tool cannot find the options corresponding to *"all other modes."* In fact, even humans could hardly find out the options corresponding to these *"modes."* For example, we could find that the *-layout* option corresponded to the physical layout mode only when we combined its name with the description of the *-table* option, *"Table mode is similar to physical layout mode."* Among the 11 false positives and 30 false negatives of conflicts, 5 and 32 were due to failure to identify R-sentences and implicitly declared conflicts, respectively (mentioned in Section 5.2). Two were due to the inability to find the corresponding options. Another two were due to the incorrect extraction because of a lack of human common sense. The *-RSAPublicKey_in* and *-RSAPublicKey_out* options were put together, sharing the description *"Like -pubin and -pubout except RSAPublicKey format is used instead."* Since the subject of the description was not plural, our tool mistakenly treated these two options as the same option, resulting in a false positive (i.e., *-RSAPublicKey_in* versus *-pubout*) and a false negative (i.e., *-RSAPublicKey_out* versus *-pubout*).

5.4 Effectiveness of Prioritization Technique (RQ4)

To evaluate the effectiveness of our prioritization technique mentioned in Section 3.5, we randomly selected ten programs from our real-world dataset and provided each program with five shuffled pruned valid combination lists (CarpetFuzz-Rand) to compare with the prioritized combinations. Each combination list was fuzzed for 48 hours. We then took the average number of unique paths $\bar{n}$ as the evaluation criterion, defined as,

$$\bar{n} = \frac{\sum_{i=1}^{5} |E_{f_i} - E_{f_i} \bigcap E_b|}{5} \tag{3}$$

where E_b represents the union of the baseline's (AFL, AFLfast, MOPT, and AFL++) edge coverage in five runs in Section 5.1, which stands for their coverage capability. E_{f_i} represents the edge coverage of CarpetFuzz and CarpetFuzz-Rand in the i-th run. The results are shown in Table 2. With our prioritization technique, CarpetFuzz found more edges on each program that other fuzzers could not discover (7% more on average), proving that this technique was effective in finding more unique edges. In three programs, the improvement was more evident (more than 10%). Especially in *openssl-ec*, CarpetFuzz found 20% more unique edges than CarpetFuzz-RAND. In seven programs, the capability of finding unique edges was slightly improved (within 5%).

Interesting findings When analyzing those combinations with the lowest priority, we found that some conflicts were only declared at runtime. For example, there were four conflicts in *img2sixel* (*-P* versus *-8*, *-p* versus *-e*, *-p* versus *-I*, and *-p* versus *-b*) not declared in any online documentation, manpage, or help command. We could get a hint only when running the program with any of these combinations, like *"option -p, –colors conflicts with -I, –high-color."* We also found that two dependencies in *tiffcrop* were declared conflicts at

Table 2: Unique edges of CarpetFuzz with and without prioritization. ALL: the union of AFL, AFLfast, MOPT, and AFL++. Ranked: CarpetFuzz. Random: CarpetFuzz-RAND.

Program	**Compare to AFL**			**Compare to ALL**		
	Ranked	Random	Rate	Ranked	Random	Rate
cmark	**920.8**	916.6	1.00	**871.6**	867	1.01
editcap	**480.8**	480	1.00	**480.4**	479.4	1.00
img2sixel	**492.8**	485	1.02	**484.6**	427.4	1.13
jpegoptiom	**117.4**	114.4	1.03	**117.4**	114.4	1.03
openssl-asn1parse	**480.8**	480	1.00	**480.4**	479.4	1.00
openssl-ec	**1847**	1541.2	1.20	**1823.6**	1521.2	1.20
pdftotext	**60.8**	60.2	1.01	60	60	1.00
speexdec	**98.8**	98.2	1.01	**98.8**	98.2	1.01
tiffcp	**660.6**	534	1.24	**485**	376.8	1.29
tiffcrop	**838**	811.2	1.03	**739.2**	727.8	1.02
Average	**599.78**	552.08	1.05	**564.1**	515.16	1.07

runtime (*-K* versus *-S*, *-J* versus *-S*). We could find a comment left by the author in the source code, *"Maybe someday but not for now,"*, which implied that the dependencies in the documentation exist but were not implemented yet. Although these invalid combinations were missed in the previous steps, our prioritization technique still found them by ranking the dry run's coverage and gave them the lowest priority.

5.5 Fuzzing performance compared to the state-of-the-art techniques (RQ5)

In order to evaluate whether extracting the relationships among options could improve the fuzzing performance, we selected the SOTA option configuration fuzzing tool, POWER [29], for comparison. POWER is also a fuzzer supporting switching option combinations on the fly. Still, the combinations it used were generated from random mutation (insert, remove, replace), unlike CarpetFuzz. We evaluated CarpetFuzz [6] and POWER on the POWER's benchmark (same programs, same versions, same seeds). Each program in the benchmark was fuzzed by CarpetFuzz and POWER for 24 hours, repeated 5 times. For crashes reported during fuzzing, we used the top three entries of the stack traces from AddressSanitizer [43] for deduplication, as suggested in [26]. From these programs, CarpetFuzz extracted 225 relationships with a precision of 90.67% and found 94 unique crashes after deduplication, 1.71 times POWER (55), as shown in Table 3. Among the 94 unique crashes, 70 were not discovered by POWER, demonstrating the advantage of CarpetFuzz in finding vulnerabilities. Although it is also possible for POWER to generate corresponding combinations through mutation, the time required may be very long due to randomness. In contrast, CarpetFuzz can generate these combinations quickly based on the extracted relationship, demonstrating the importance of considering relationships.

Note that POWER also found 31 unique crashes that CarpetFuzz did not discover. We acknowledged that random mu-

[6]Since POWER is based on AFL++, we used the AFL++-based CarpetFuzz for comparison, to be fair.

Table 3: Vulnerabilities discovered on the POWER's dataset. Exclusive crash: crash only found by the specific fuzzer. Average edge: average number of edges across all test cases.

Program	**CarpetFuzz**			**POWER**		
	#Uniq. crash	**#Excl. crash**	**Avg. edge**	**#Uniq. crash**	**#Excl. crash**	**Avg. edge**
avconv	**3**	**3**	**2623.73**	0	0	2209.15
bison	**5**	**3**	**1365.97**	4	2	1148.31
cflow	3	1	479.10	3	1	**642.79**
cjpeg	0	0	**115.58**	0	0	73.46
djpeg	0	0	**132.33**	0	0	52.97
dwarfdump	1	0	**1857.72**	**2**	**1**	1376.34
exiv2	**2**	**1**	1078.32	1	0	**1402.15**
ffmpeg	1	0	2134.40	**2**	**1**	**2768.37**
gm	0	0	**1172.00**	0	0	566.98
gs	0	0	9306.35	0	0	**9447.74**
jasper	0	0	**741.13**	0	0	524.47
mpg123	**4**	**4**	**921.51**	0	0	421.19
mutool	0	0	**52.44**	0	0	48.33
nasm	4	4	**936.64**	**9**	**9**	898.18
objdump	**10**	**9**	**1111.69**	2	1	1012.09
pdftohtml	0	0	**1672.19**	0	0	1536.17
pdftopng	4	1	**1683.99**	**5**	**2**	1592.40
pdftops	6	1	**1552.22**	6	1	1528.50
pngfix	0	0	205.91	0	0	**210.00**
pspp	**28**	**21**	**1080.37**	12	5	1021.12
readelf	**3**	**3**	**464.49**	0	0	460.36
size	0	0	309.68	0	0	**428.10**
tiff2pdf	0	0	**684.29**	0	0	661.90
tiff2ps	0	0	**377.32**	0	0	376.28
tiffinfo	0	0	**367.90**	0	0	331.87
vim	1	0	**4733.48**	**2**	**1**	4279.16
xmlcatalog	0	0	**768.18**	**4**	**4**	748.26
xmllint	**14**	**14**	1155.51	3	3	**1467.75**
xmlwf	0	0	134.90	0	0	**160.57**
yara	**5**	**5**	**434.81**	0	0	407.62
Total	**94**	**70**		55	31	

tation may be more effective in finding unexpected behaviors caused by some invalid combinations. However, CarpetFuzz aims to explore the deep code related to option combinations, which requires the tested combination to be valid. We analyzed the average unique edges triggered by all test cases generated by CarpetFuzz and POWER, as shown in columns 3 and 6 of Table 3. In 22 programs, the average number of edges of CarpetFuzz was greater than that of POWER. In 8 programs, CarpetFuzz performed worse than POWER. The results for seven programs (except *ffmpeg*) were caused by treating the strings inserted in the command line by POWER as input file names, which brought in more edges. Note that these "new" input files would not be mutated as fuzzers only mutates files symbolized as "@@." So the new edges they brought would not be used to find new bugs, which is why CarpetFuzz still found more bugs in these programs even with fewer average edges (19 versus 7). The result for the rest one (i.e., *ffmpeg*) was due to a lack of supporting position-specific options, which only take effect in a specific position. Since CarpetFuzz currently does not support constraints related to option positions, it cannot generate combinations that allow these options to take effect and thus found fewer edges and bugs (1 versus 2) in this program. Note that programs with such options are rare (less than 5%). More detailed analyses of these eight programs' results are presented in Appendix G.

The results showed that the average unique edges of test cases generated by CarpetFuzz were 1.16 times that of POWER on average, up to 2.50 times in *djpeg*, proving that CarpetFuzz is more effective in exploring option-related deep code.

5.6 Real-world Vulnerabilities Discovered (RQ6)

We have continuously run the CarpetFuzz-assisted fuzzer on our real world dataset to discover new vulnerabilities. So far, we have found 57 unique crashes, of which 43 are 0-days, as shown in Table 4. These vulnerabilities cover a variety of vulnerability types, including double free, use after free, buffer overflow, floating point exception, and assertion failures. In these 43 0-days, we have obtained 30 CVE IDs, and the other 13 are still under request. We analyzed the triggering conditions of these vulnerabilities and calculated the minimum number of options required to trigger them (i.e., the shortest combination). Among these vulnerabilities, 47 require at least one option to be triggered (82.46%), and 30 of them require at least two options (52.63%), proving the importance of options in vulnerability discovery. We found that for these vulnerabilities, combining all of the options may not be helpful due to conflicts among options. For example, the shortest combination to trigger the heap buffer overflow in *tiffcp.c:1373* (CVE-2022-0924) is *"-i -s -p separate."* Since the *-s* option conflicts with the *-t* option, any combination containing both these two options will not trigger this vulnerability, including the combination of all options. Similarly, the *-Z* option in *tiffcrop* conflicts with the *-X*, *-Y*, and *-z* options, and the *-p* option in *img2sixel* conflicts with *-e*, *-I*, and *-b* options, so combining all options would not trigger these programs' vulnerabilities, such as CVE-2022-2058 and CVE-2022-29978. We analyzed the longest combinations of these three vulnerabilities, which were 14 (18 in total), 25 (33 in total), and 23 (28 in total), respectively, demonstrating that these vulnerabilities could not be triggered by combining all options. It is worth noting that most of the programs have been integrated into the OSS-Fuzz project [19] and have been extensive continuous fuzzing in recent years. However, we can still find new vulnerabilities in these programs, proving that fuzzing with valid option combinations is still a novel and effective method for discovering vulnerabilities at this stage.

6 Discussion

Limitation and future work. Although CarpetFuzz achieved good results in relationship identification and exraction, there were still some false negatives and false positives. One reason is the limitations of the NLP model (spaCy, NLTK, and AllenNLP) that CarpetFuzz relies on, which mainly comes from the dependency parser's accuracy. Although the accuracy has reached 97% [46] and is enough for fuzzing, we find that parsing errors usually occur when dealing with complex sentences or computer terms, causing seven false negatives in Section 5.2. Note that a few false negatives will only affect the number of combinations (i.e., more invalid combinations) to test but not the final fuzzing results since our prioritization technique will give these missed invalid combinations the lowest priority (Section 3.5). Our performance will be further improved with NLP's development. Another reason is a lack of human common sense, as mentioned in Section 5.2. Some conflicts can only be determined with certain common sense, such as "horizontal dimension" and "vertical dimension." Although they are both "dimensions," humans know that "vertical" and "horizontal" are two non-interfering with each other. In future work, we consider adding a commonsense knowledge graph to assist CarpetFuzz in identifying such relationships. In addition to being found from descriptions, relationships can also be determined by comparing option names, as mentioned in Section 5.2. CarpetFuzz currently identifies relationships based on descriptions and may perform better by combining description and option name.

Since CarpetFuzz is implemented based on AFL, it also has the limitations of AFL itself. For example, CarpetFuzz only detects vulnerabilities that crash the programs. However, by compiling the target programs with LLVM Sanitizers, such as TSan [51] and UBSan [52], the fuzzer in CarpetFuzz can also detect non-crash vulnerabilities. Additionally, CarpetFuzz only runs on Linux systems. This limitation can be solved by implementing CarpetFuzz to fuzzers supporting other systems, such as WINAFL [18]. Since other systems may use different identifiers to represent options (e.g., "/" in Windows), the code for parsing documents in CarpetFuzz must also be changed accordingly.

Importance of considering a large number of combinations. Considering a large number of combinations is important for finding potential option-related vulnerabilities. In Section 5.6, more than half of vulnerabilities require specific combinations to be triggered, which only accounts for a small proportion of all combinations (e.g., 1.56% for CVE-2022-3626). Moreover, due to the conflicts among options, combining all options would cause the programs to throw exceptions and thus not help discover these vulnerabilities. Without considering a large number of combinations, these very few specific combinations will be missed, causing these vulnerabilities to be missed.

Importance of filtering invalid option combinations. Filtering the invalid option combinations is important for saving time in finding vulnerability-related combinations. In Section 5.6, CarpetFuzz found 18 vulnerabilities that require four or more options to be triggered, accounting for 31.57% of all discovered vulnerabilities. CarpetFuzz can save 85.06% of the time by filtering invalid combinations for vulnerabilities with four or more options. Although most vulnerabilities (68.43%) in Table 4 have fewer than four options, the number of combinations is still large (e.g., 6850 for *jpegoptim*) due to the large

Table 4: Vulnerabilities discovered by CarpetFuzz. #Options: minimum number of options to trigger the vulnerability.

Program	**Vulnerability Type**	**Location**	**0-day or 1-day?**		**Options**		**Fixed?**
			0-day?	**CVE**	**Require?**	**#Options**	
img2sixel	segmentation violation	fromgif.c:283	NO	N/A	YES	1	NO
	assertion failure	stb_image.h:1894	YES	CVE-2022-29977	NO	0	NO
	floating point exception	encoder.c:633	YES	CVE-2022-29978	YES	1	NO
		encoder.c:636	YES	Requesting	YES	1	NO
	heap-buffer-overflow	quant.c:876	YES	Requesting	YES	2	NO
jpegoptim	segmentation violation	jpegoptim.c:711	NO	N/A	YES	2	YES
		jpegoptim.c:906	YES	Requesting	YES	1	YES
		jpegoptim.c:632	YES	Requesting	NO	0	YES
		jpegoptim.c:1055	YES	Requesting	YES	1	YES
jpegtran	floating point exception	transupp.c:157	YES	Requesting	YES	2	YES
		transupp.c:161	YES	Requesting	YES	2	YES
jq	heap-use-after-free	jv.c:31	NO	N/A	YES	4	YES
	assertion failure	jv_parse.c:305	NO	N/A	YES	1	NO
openssl-asn1parse	double-free	mem.c:277	YES	CVE-2022-4450	YES	1	YES
openssl-rsa	assertion failure	packet.c:387	YES	Requesting	NO	0	YES
pdftops	segmentation violation	gmem.cc:356	YES	Requesting	YES	1	NO
	stack-buffer-overflow	gmem.cc:148	NO	N/A	NO	0	NO
pdftotext	segmentation violation	gmem.cc:356	YES	Requesting	NO	0	NO
	stack-buffer-overflow	Dict.cc:98	NO	N/A	NO	0	NO
tcpprep	assertion failure	tree.c:746	NO	N/A	YES	1	YES
		tree.c:746	YES	Requesting	YES	1	YES
tcpreplay	assertion failure	send_packets.c:116	YES	Requesting	YES	5	NO
tiffcp	heap-buffer-overflow	tiffcp.c:1373	YES	CVE-2022-0924	YES	3	YES
		tiffcp.c:948	YES	CVE-2022-4645	NO	0	YES
	assertion failure	tif_read.c:99	YES	CVE-2022-0865	NO	0	YES
	segmentation violation	tif_lzw.c:619	YES	CVE-2022-1622	YES	1	YES
		tif_lzw.c:624	YES	CVE-2022-1623	YES	1	YES
tiffcrop	heap-buffer-overflow	tif_unix.c:340	YES	CVE-2022-3626	YES	6	YES
		tif_unix.c:346	YES	CVE-2022-1056	YES	1	YES
		tif_unix.c:346	YES	CVE-2022-3597	YES	4	YES
		tif_unix.c:346	YES	CVE-2022-3627	YES	4	YES
		tif_unix.c:368	YES	CVE-2023-0801	YES	3	YES
		tiffcrop.c:3414	NO	N/A	YES	6	YES
		tiffcrop.c:3502	YES	CVE-2023-0800	YES	2	YES
		tiffcrop.c:3516	YES	CVE-2023-0803	YES	4	YES
		tiffcrop.c:3604	YES	CVE-2022-3598	YES	1	YES
		tiffcrop.c:3609	YES	CVE-2023-0804	YES	4	YES
		tiffcrop.c:3724	YES	CVE-2023-0802	YES	3	YES
		tiffcrop.c:6905	YES	CVE-2022-2953	YES	5	YES
		tiffcrop.c:7345	YES	CVE-2022-3599	YES	1	YES
		tiffcrop.c:8023	NO	N/A	NO	0	YES
		tiffcrop.c:8903	NO	N/A	YES	3	YES
	segmentation violation	tif_unix.c:340	YES	CVE-2022-0907	YES	1	YES
		tif_unix.c:368	YES	CVE-2023-0797	YES	4	YES
		tiffcrop.c:3400	YES	CVE-2023-0798	YES	4	YES
		tiffcrop.c:3488	YES	CVE-2023-0795	YES	4	YES
		tiffcrop.c:3592	YES	CVE-2023-0796	YES	4	YES
		tiffcrop.c:6247	YES	Requesting	YES	6	NO
	floating point exception	tiffcrop.c:5801	NO	N/A	YES	1	YES
		tiffcrop.c:5802	YES	CVE-2022-0909	NO	0	YES
		tiffcrop.c:5807	NO	N/A	YES	4	YES
		tiffcrop.c:5808	NO	N/A	YES	4	YES
		tiffcrop.c:5817	YES	CVE-2022-2056	YES	3	YES
		tiffcrop.c:5818	NO	N/A	YES	3	YES
		tiffcrop.c:5936	YES	CVE-2022-2057	YES	2	YES
		tiffcrop.c:5941	YES	CVE-2022-2058	YES	5	YES
	heap-use-after-free	tiffcrop.c:3701	YES	CVE-2023-0799	YES	4	YES
Total		57	43	30	47		45

number of options, and CarpetFuzz can save 20.71% of the time. Additionally, CarpetFuzz found 1.71 times more unique crashes than POWER (Section 5.5), which generates combinations through random mutation without filtering invalid combinations, proving that filtering invalid option combinations is important in new vulnerability discovery.

Optimal number of options. CarpetFuzz used the N-wise method to prune valid option combinations. The smaller the value of N, the better the effect of pruning, but the interaction of more than N options will be ignored. The choice of 6 in this paper is based on the research that almost all flaws are caused by the interaction of no more than six factors [27]. Moreover, three of the 57 crashes need at least 6 options to be triggered, which would be missed with $N < 6$, demonstrating the necessity of considering 6-wise combinations. We are unsure if crashes requiring more options exist. Considering the effect of increasing the value of N on the number of combinations (e.g., 679 for $N = 6$ and 1658 for $N = 7$ in *tiffcrop*), we think $N = 6$ is an appropriate choice in practice.

7 Conclusion

We designed and implemented CarpetFuzz, an NLP-based fuzzing assistance technique for extracting program option constraints. Benefitting from active learning, machine learning, and NLP techniques, CarpetFuzz accurately extracted relationships among options from documents and filtered out 67.91% of option combinations. With the pruned valid combinations, CarpetFuzz helped AFL find 45.97% more paths that other fuzzers cannot discover on 20 popular programs and found 57 unique crashes, of which 30 were assigned CVE IDs. Also, CarpetFuzz found 94 unique crashes on the previous work's benchmark, 1.71 times that of the previous work.

Acknowledgement

We are grateful to our shepherd for his valuable guidance and assistance, and thank all the anonymous reviewers for their insightful comments. The authors are supported in part by NSFC (U1836211, 92270204), Beijing Natural Science Foundation (No.M22004), Youth Innovation Promotion Association CAS, Beijing Academy of Artificial Intelligence (BAAI) and a research grant from Huawei.

References

[1] 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)*, pages 1597–1612. IEEE, 2020.

[2] bjascob. bjascob/lemminflect: A python module for english lemmatization and inflection. https://github.com/bjascob/LemmInflect, 2022.

[3] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 678–689, 2020.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[6] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[7] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[8] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of jvm implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–99, 2016.

[9] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 736–747. IEEE, 2019.

[10] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[11] Ido Dagan and Sean P Engelson. Committee-based sampling for training probabilistic classifiers. In *Machine Learning Proceedings 1995*, pages 150–157. Elsevier, 1995.

[12] Debian. Debian manpages. https://manpages.debian.org, 2022.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[14] 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.

[15] 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, 2020.

[16] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[17] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th international symposium on software testing and analysis*, pages 213–224, 2016.

[18] Google. A fork of afl for fuzzing windows binaries. https://github.com/googleprojectzero/winafl, 2022.

[19] Google. Oss-fuzz. https://google.github.io/oss-fuzz/, 2022.

[20] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 230–243, 2021.

[21] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[22] Sabine Iatridou and Hedde Zeijlstra. Negation, polarity, and deontic modals. *Linguistic inquiry*, 44(4):529–568, 2013.

[23] ImageMagick. Imagemagick - command-line tools: Magick. https://imagemagick.org/script/magick.php, 2022.

[24] Allen Institute. Allennlp - allen institute for ai. https://allenai.org/allennlp, 2022.

[25] Michael Israel. The pragmatics of polarity. *The handbook of pragmatics*, 16:701–723, 2004.

[26] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[27] Rick Kuhn. Practical applications of combinatorial testing, 2012.

[28] Steven M LaValle, Michael S Branicky, and Stephen R Lindemann. On the relationship between classical grid search and probabilistic roadmaps. *The International Journal of Robotics Research*, 23(7-8):673–692, 2004.

[29] Ahcheong Lee, Irfan Ariq, Yunho Kim, and Moonzoo Kim. Power: Program option-aware fuzzer for high bug detection ability. In *the 15th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2022.

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

[31] Tao Lv, Ruishi Li, Yi Yang, Kai Chen, Xiaojing Liao, XiaoFeng Wang, Peiwei Hu, and Luyi Xing. Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1837–1852, 2020.

[32] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, 2019.

[33] Valentin JM 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*, pages 1024–1036, 2020.

[34] Microsoft. Pict. https://github.com/microsoft/pict, 2022.

[35] MozillaSecurity. Fuzzing resources for feeding various fuzzers with input. https://github.com/MozillaSecurity/fuzzdata, 2022.

[36] openhub. The apache http server open source project on open hub. https://www.openhub.net/p/apache, 2022.

[37] openhub. The mysql open source project on open hub. https://www.openhub.net/p/mysql, 2022.

[38] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language api descriptions. In *2012 34th international conference on software engineering (ICSE)*, pages 815–825. IEEE, 2012.

[39] NLTK Project. Nltk: Natural language toolkit. https://www.nltk.org/index.html, 2022.

[40] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[41] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 861–875, 2014.

[42] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. http://is.muni.cz/publication/884893/en.

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

[44] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. Mtfuzz: fuzzing with a multi-task neural network. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 737–749, 2020.

[45] Suhwan Song, Chengyu Song, Yeongjin Jang, and Byoungyoung Lee. Crfuzz: Fuzzing multi-purpose programs through input validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 690–700, 2020.

[46] spaCy. Facts & figures. https://spacy.io/usage/facts-figures, 2022.

[47] spaCy. Industrial-strength natural language processing in python. https://spacy.io, 2022.

[48] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* icomment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 145–158, 2007.

[49] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. acomment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 11–20. IEEE, 2011.

[50] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 260–269. IEEE, 2012.

[51] The Clang Team. Threadsanitizer. https://clang.llvm.org/docs/ThreadSanitizer.html, 2022.

[52] The Clang Team. Undefinedbehaviorsanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2022.

[53] The Clang Team. Writing an llvm pass. https://llvm.org/docs/WritingAnLLVMPass.html, 2022.

[54] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *NDSS*, 2021.

[55] Zi Wang, Ben Liblit, and Thomas Reps. Tofu: Target-oriented fuzzer. *arXiv preprint arXiv:2004.14375*, 2020.

[56] Bernd Warken. groff - debian bullseye - debian manpages. https://manpages.debian.org/bullseye/groff/groff.7.en.html, 2021.

[57] Wikipedia. N-wise testing. https://en.wikipedia.org/wiki/All-pairs_testing#N-wise_testing, 2021.

[58] Wikipedia. Command-line interface. https://en.wikipedia.org/wiki/Command-line_interface, 2022.

[59] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. Dase: Document-assisted symbolic execution for improving automated software testing. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 620–631. IEEE, 2015.

[60] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael Godfrey. Leveraging documentation to test deep learning library functions. *arXiv preprint arXiv:2109.01002*, 2021.

[61] 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, 2020.

[62] Michal Zalewski. afl-showmap. https://manpages.debian.org/unstable/afl++/afl-showmap.8.en.html, 2022.

[63] Michal Zalewski. american fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2022.

[64] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Testing configurations. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2022. Retrieved 2022-05-18 12:31:52+02:00.

[65] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. C2s: translating natural language comments to formal program specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 25–37, 2020.

[66] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. Registered report: Fuzzing configurations of program options. In *1st International Fuzzing Workshop (FUZZING'22)*, 2022.

[67] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.

[68] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language api documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 307–318. IEEE, 2009.

[69] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Computing Surveys (CSUR)*, 2022.

Appendix

A Category of the explicitly declared relationships in this paper

Table 5 shows the categories of the explicitly declared relationships in this paper.

Table 5: Category of the explicitly declared relationships in this paper.

Category	Example
Conflict	The -level1 option cannot be used with -form.
Dependency	Make sense only when -p option is specified.
Implication	This flag implies (and is incompatible with) -e.
Similarity	Like -a but write only the image dimensions.
Supersedence	-G supersedes usage of any -l or -m options.

B Algorithm of the active learning

Algorithm 1 shows the active learning algorithm framework.

Algorithm 1 The active learning algorithm framework.

```
Input: An unlabeled dataset, $U = \{x_1, x_2, ..., x_n\}$, $n$ denotes the number of samples in $U$;
    A small labeled dataset based on keywords, $L = \{x_1, x_2, ..., x_m\}$, $m$ denotes the number of samples in $L$;
    Oracle $f : \mathcal{X} \leftarrow \mathcal{Y}$
    The number of samples to select, $k$;
Output: Final classifier $\mathcal{C}$
    Initialization: Unlabeled data $\mathcal{U} := U$, labeled data $\mathcal{L} := L$, Classifier $C$ based on $\mathcal{L}$;
    for $t = 1 \rightarrow T$ do
        $Entropy(x^{(i)}) \leftarrow$ calculate entropy on $\mathcal{U}$ using classifier $\mathcal{C}$;
        Select top $k$ $x^{(i)} \leftarrow \arg\max_{x \in \mathcal{U}} Entropy(x)$
        Evaluate $y^{(i)} \leftarrow f(x^{(i)})$
        Update $\mathcal{U} \leftarrow \mathcal{U} \backslash x^{(i)}$
        Update $\mathcal{L} \leftarrow \mathcal{L} \cup (x^{(i)}, y^{(i)})$
        Update $\mathcal{C} \leftarrow \mathcal{C}(\mathcal{L})$
    return $\mathcal{C}$
```

C Keywords used in this paper

The keywords used in this paper is shown in Table 6.

Table 6: Keywords for relationship extraction.

Category	Initial state	Keywords
Conflict	S_C	implicit, conflict, discard, exclude, cancel, discourage, disregard, alias, terminate, imply, override, equivalent, ignore, similar, include, identical, better, exclusive, supersede, superfluous, same, replace, obsolete
Conflict	S_N	disable, unspecified, unprovide, unavailable, separate
Dependency	S_D	support, match, require, need, valid, active, affect, assume
Dependency	S_N	use, enable, specify, give, provide, meaningful, available, combine, appear

D Real world dataset

Table 7 shows the 20 programs in our real world dataset.

Table 7: Target programs evaluated in our experiments. OPT.: option, REL.: relationship.

Program	Package	Format	#OPT.	#REL.
cmark	cmark-git-9c8e8	md	9	3
editcap	wireshark-4.0.1	pcap	31	4
eu-elfclassify	elfutils-0.188	elf	22	71
img2sixel	libsixel-git-6a5be	jpg	28	6
jpegoptim	jpegoptim-1.5.0	jpg	34	29
jpegtran	libjpeg-turbo-2.1.4	jpg	21	1
jq	jq-1.6	json	24	4
lrzip	lrzip-0.651	lrz	22	10
ogg123	vorbis-tools-1.4.2	ogg	13	1
openssl-asn1parse	openSSL-git-31ff3	pem	12	2
openssl-ec	openSSL-git-31ff3	pem	16	3
openssl-rsa	openSSL-git-31ff3	pem	28	71
pdftops	xpdf-4.0.3	pdf	30	30
pdftotext	xpdf-4.0.3	pdf	24	5
podofoencrypt	podofo-0.9.8	pdf	11	1
speexdec	speex-1.2.1	spx	12	18
tcpprep	tcpreplay-4.4.2	pcap	19	17
tcpreplay	tcpreplay-4.4.2	pcap	29	20
tiffcp	libtiff-git-b51bb	tiff	18	4
tiffcrop	libtiff-git-b51bb	tiff	33	6

E Filtered option combinations

The result of filtering invalid option combinations is shown in Table 8.

F Comparison between 6-wise and random sampling

Table 9 shows the result of pruning.

G In-depth analysis of the performance in Section 5.5

We have analyzed these eight programs and their manuals manually. The results for seven programs (except ffmpeg) were caused by treating the strings inserted in the command line by POWER as input file names. The result for the rest one (i.e., ffmpeg) was due to a lack of supporting position-specific options. We also found that two programs' documents did not

Table 8: Filtered option combinations based on the extracted relationships. REL.: number of relationships extracted by CarpetFuzz , PRC.: precision, PCT.: percentage of the filtered option combinations.

Program	#REL.	Invalid Combination		
		PCT.	PRC.	Recall
cmark	2	43.75%	100.00%	82.35%
editcap	4	62.50%	100.00%	100.00%
eu-elfclassify	66	99.68%	100.00%	99.80%
img2sixel	5	69.24%	100.00%	87.10%
jpegoptim	22	94.99%	100.00%	97.54%
jpegtran	1	25.00%	100.00%	100.00%
jq	4	68.36%	100.00%	100.00%
lrzip	6	68.75%	100.00%	84.62%
ogg123	1	25.00%	100.00%	100.00%
openssl-asn1parse	2	43.75%	100.00%	100.00%
openssl-ec	3	50.00%	100.00%	100.00%
openssl-rsa	69	99.85%	99.99%	99.93%
pdftops	31	98.10%	99.35%	100.00%
pdftotext	9	85.35%	68.46%	90.12%
podofoencrypt	1	25.00%	100.00%	100.00%
speexdec	12	88.28%	100.00%	93.39%
tcpprep	17	98.85%	100.00%	100.00%
tcpreplay	21	99.62%	99.71%	100.00%
tiffcp	2	43.75%	100.00%	70.00%
tiffcrop	4	68.36%	92.77%	79.05%
Average		67.91%	98.01%	94.19%

Table 9: Result of pruning. OC: option combination.

Program	6-wise		Sampling	
	#OC	#Edge	#OC	#Edge
cmark	150	1,086	288	1,086
editcap	704	991	100,000	993
eu-elfclassify	1,110	111	26,624	111
img2sixel	610	1,211	100,000	1,244
jpegoptim	988	208	100,000	213
jpegtran	414	1,833	100,000	2,076
jq	558	1,232	100,000	1,354
lrzip	645	113	100,000	114
ogg123	257	351	6,144	351
openssl-asn1parse	249	2,212	2,304	2,212
openssl-ec	385	6,266	32,768	6,266
openssl-rsa	2,042	5,592	100,000	5,595
pdftops	1,494	608	100,000	609
pdftotext	786	562	100,000	562
podofoencrypt	203	237	1,536	237
speexdec	271	573	480	577
tcpprep	710	573	6,048	573
tcpreplay	1,327	565	100,000	570
tiffcp	384	1,246	100,000	1,291
tiffcrop	679	1,351	100,000	1,589
Average	698.30	1,346.05	63809.60	1,381.15

clearly write all the relationships. We described the details as follows.

First, we found seven of these programs treated all non-option-related strings in the command line as input files. Since POWER randomly inserted strings into the command line during the mutation, these newly inserted strings were recognized as part of input files. For example, both the *"abc"* and *"@@"* in the command, *"cflow -a -T abc @@ -o /tmp/foo,"* will be recognized as input files and processed. If the files represented by the new strings do not exist, the new edges would be only a little bit (less than five). However, POWER often replaces the output file names with the above strings during the mutation process, causing such files to exist and more new edges. For example, POWER can replace the command *"cflow -T @@ -o /tmp/foo"* with *"cflow -a -T @@ -o abc,"* which generates the file named *"abc."* By parsing the file, the command *"cflow -a -T abc @@ -o /tmp/foo"* will have 62 new edges (even more with more such files), but no new bug was generated.

Additionally, sometimes, POWER could insert options before the program, like *"-a -T cflow @@ -o /tmp/foo."* Due to POWER's implementation, argv[0] would not be changed, and the command would be parsed as *"cflow -T cflow @@ -o /tmp/foo"* (ignoring the first option *"-a"*). The program would treat the second *"cflow"* as an input and parse itself. If the input type of the program happened to be ELF (e.g., size), the new edges could be a lot (e.g., 92).

Second, we found some options in *ffmpeg* only take effect in a specific position. For example, the *"-f"* option would specify the output format only between the input and output in the command line. Since CarpetFuzz currently does not support constraints related to option positions, it cannot generate combinations that allow *-f* to take effect, resulting in 1208 fewer edges.

Third, after analyzing those combinations with the lowest priority, we found two programs' documents do not mention some conflicts. Specifically, the *cflow*'s document missed the conflict between *-format=posix* and *–emacs*, and the *exiv2*'s document missed the conflicts among *-D*, *-P*, *-t*, *-p*, *-t*, and *-d* options. Since our prioritization technique gave the missed invalid combinations the lowest priority, as mentioned in Section 3.5, the impact was little.