CrFuzz: Fuzzing Multi-purpose Programs through Input Validation

Suhwan Song
Seoul National University
sshkeb96@snu.ac.kr

Chengyu Song
UC Riverside
csong@cs.ucr.edu

Yeongjin Jang
Oregon State University
yeongjin.jang@oregonstate.edu

Byoungyoung Lee*
Seoul National University
byoungyoung@snu.ac.kr

ABSTRACT

Fuzz testing has been proved its effectiveness in discovering software vulnerabilities. Empowered its randomness nature along with a coverage-guiding feature, fuzzing has been identified a vast number of vulnerabilities in real-world programs. This paper begins with an observation that the design of the current state-of-the-art fuzzers is not well suited for a particular (but yet important) set of software programs. Specifically, current fuzzers have limitations in fuzzing programs serving multiple purposes, where each purpose is controlled by extra options.

This paper proposes CrFuzz, which overcomes this limitation. CrFuzz designs a clustering analysis to automatically predict if a newly given input would be accepted or not by a target program. Exploiting this prediction capability, CrFuzz is designed to efficiently explore the programs with multiple purposes. We employed CrFuzz for three state-of-the-art fuzzers, AFL, QSYM, and MOpt, and CrFuzz-augmented versions have shown 19.3% and 5.68% better path and edge coverage on average. More importantly, during two weeks of long-running experiments, CrFuzz discovered 277 previously unknown vulnerabilities where 212 of those are already confirmed and fixed by the respected vendors. We would like to emphasize that many of these vulnerabilities were discoverd from FFMpeg, ImageMagick, and Graphicsmagick, all of which are targets of Google's OSS-Fuzz project and thus heavily fuzzed for last three years by far. Nevertheless, CrFuzz identified a remarkable number of vulnerabilities, demonstrating its effectiveness of vulnerability finding capability.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**.

KEYWORDS

Fuzz testing; Coverage-guided fuzzing;

*Corresponding author

ACM Reference Format:
Suhwan Song, Chengyu Song, Yeongjin Jang, and Byoungyoung Lee. 2020. CrFuzz: Fuzzing Multi-purpose Programs through Input Validation. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3368089.3409769

1 INTRODUCTION

The fuzz testing is a popular technique to discover software vulnerabilities. It keeps generating a random input and testing a target program if the program exhibits a violation behavior, such as memory access violation or triggering safety assertions. One notable feature that most state-of-the-art fuzzers are employing today is a coverage-guided fuzzing feature. Under the assumption that more testing coverage would yield better vulnerability detection capability, this feature keeps tracking of execution coverage at runtime and provides coverage feedback to guide the fuzzing procedure (e.g., guiding how the fuzzer generates, mutates, or selects an input). Indeed, fuzz testing has been proved its effectiveness in discovering vulnerabilities in a vast number of real-world software [2, 4, 6, 13, 24, 32, 39].

This research started with a research question whether current state-of-the-art fuzzers are well designed for all different set of target programs. In particular, we observe that current fuzzers are not suited to fuzz programs taking various command line options, which we call multi-purpose programs. Multi-purpose programs serve multiple operational behaviors (e.g., in the case of `FFmpeg`, it supports encoding/decoding with a large number of codecs as well as many different filtering features), and the command line option is used to specify which operational behavior would be activated at runtime. Since current fuzzers are not designed to cater this command line option, it is challenging to efficiently explore various operational behaviors while switching the command line options. This limitation stems from the fact that current fuzzers are designed to handle a single-dimensional input space (which is often a file input). As such, since command line options introduce an additional input space, the coverage-guided features of current fuzzers are not tailored to fuzz with such multi-dimensional input space.

In this paper, we propose CrFuzz, a fuzzing technique to specifically designed to fuzz multi-purpose programs. The key component of CrFuzz is a validity checker, which utilizes a machine learning technique, particularly a clustering analysis. The validity checker faithfully predicts if a newly provided input is a valid input (i.e., an

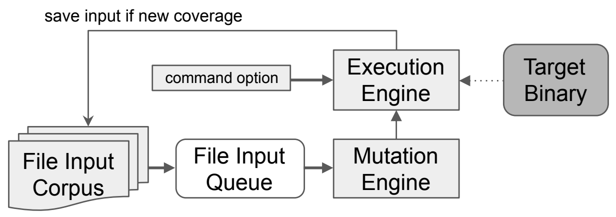


Figure 1: The workflow of general coverage-guided fuzzers

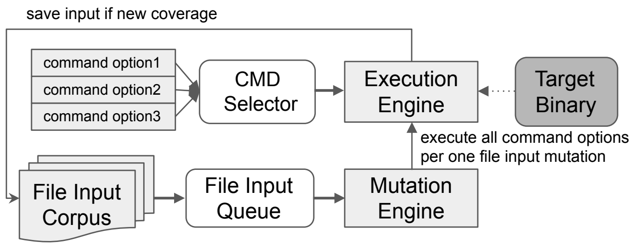


Figure 2: The workflow of naive multi-purpose program fuzzer

input supposed to be accepted by a target program) or an invalid input (i.e., an input rejected by the target program). Observing that the coverage feedback used by current fuzzers is not suited for the validity checker, CrFuzz designs new coverage feedback metric, a validity pair, which efficiently captures the differences between valid input execution and invalid input execution. Leveraging this validity checker, CrFuzz develops a multi-purpose program fuzzer, which efficiently explores many different operational behaviors.

To clearly show its effectiveness, we applied CrFuzz for three popular state-of-the-art fuzzers, AFL, QSym, and MOpt. Accordingly to our evaluation, CrFuzz has shown 19.3% and 5.68% better path coverage and edge coverage, respectively. More importantly, CrFuzz has shown outstanding performances in discovering new vulnerabilities. CrFuzz discovered 277 new vulnerabilities from popular programs (including FFmpeg, libtiff, GhostScript, libsixel, Xfig, MuPDF, ImageMagick, Graphicsmagick) where 212 of those are already been patched by the respected open source community.

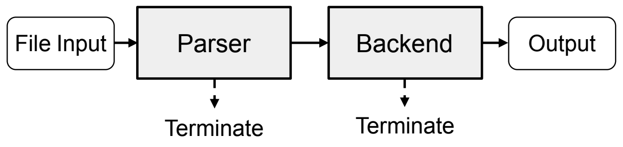
2 BACKGROUND AND MOTIVATION

In this section, we first provide necessary background of this paper (§2.1), particularly related to coverage-guided fuzzing and multi-purpose programs. Then we describe the motivation behind this paper—how to efficiently fuzz multi-purpose programs (§2.2).

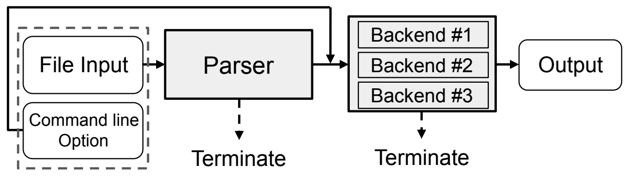
2.1 Background

Coverage-Guided Fuzzing. A coverage-guided fuzzing technique is arguably the most popular fuzzing technique to find vulnerabilities. Similar to traditional fuzzers, it keeps generating or mutating an input to execute a target program in hopes that such an input may trigger a corner case of program's logic. The key insight behind the coverage-guided fuzzing is that it keeps tracking the execution coverage of each input's execution, and leverages the coverage to mutate the input to be fuzzed next. In other words, its fuzzing process is designed to favor the input which exhibits a new coverage, such that the fuzzer can explore deep inside of program's logic.

In general, coverage-guided fuzzers work in following steps (depicted in Figure 1). First, it inserts the initial inputs (so called seeds) to the file input queue (Step 1). Then it picks one input from the file input queue (Step 2). After that, the fuzzer mutates the selected input with pre-determined mutation methods (e.g., a bit flip, performing arithmetic operations, replacing with specific integer values, or splicing) (Step 3). Then the fuzzer runs a target program with this mutated input (Step 4) while (i) checking if it triggers exception and (ii) keeping track of the execution coverage. If the mutated input visits the new coverage, that that input is inserted to the file input queue (Step 5-a). Otherwise, the mutated input will be dropped and ignored (Step 5-b). Then the fuzzer repeat this fuzzing process by returning back to Step 2.



(a) Single-purpose Program



(b) Multi-purpose Program

Figure 3: A general program logic of single-purpose programs and multi-purpose programs.

Multi-Purpose Programs. In this paper, as described in Figure 3, we categorize the programs into two types, single-purpose programs and multi-purpose programs, depending on whether a program takes command line options or not to control its multiple operational behaviors. In other words, whereas the single-purpose program simply takes a file input and does not take a command line option, the multi-purpose program takes both file input and command line options. More specifically, the multi-purpose program first attempts to parse the file input. If the given input file is invalid, the program would be terminated as it is early rejected by the parser. If valid, the program moves on to the processing stage, performing one of backend operations as instructed by the command line option.

We note that there are many multi-purpose programs as developers ship many different features within a single program binary. The representative example of multi-purpose programs would be `FFmpeg`, which is a popular video and audio converter and used as a library by a vast number of video/audio tools. Since `FFmpeg` provides many different features where each feature is being served by an individual backend engine (from encoding/decoding engines,

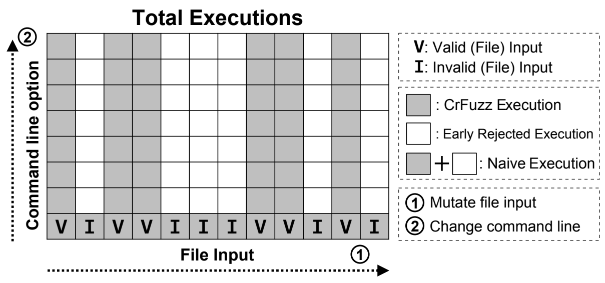


Figure 4: Input Space of Multi-purpose Program

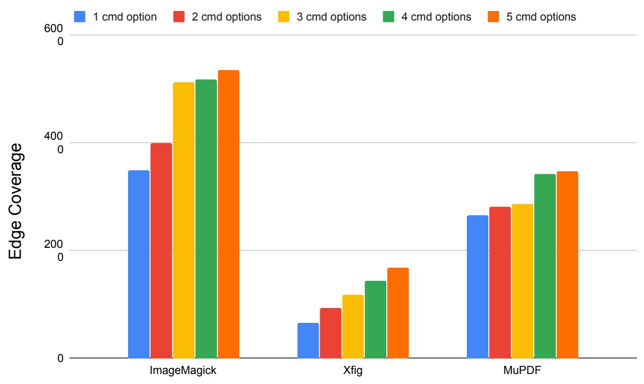


Figure 5: Measuring the edge coverage while changing the number of specified command options

dedicated backend engines for various video/audio to various filters to be applied). Another example would be `ImageMagick`, a popular image editor. It takes more than 100 different command line options to control its behavior (such as converting gif to png format), which activates various backend operations.

2.2 Motivation

Inefficient to Fuzz Multi-purpose Programs. We observe that all currently known fuzzers (including AFL, QSYM, MOpt, etc.) are not suitable to fuzz multi-purpose programs mainly due to the fact it does not cater command line options. More specifically, current fuzzers are designed to fuzz single-purpose programs, which simply simply fixes a specific command line option when running a target program. Hence, current fuzzers cannot explore various features provided by backends, critically limiting its testing capability.

One may attempt to redesign the current fuzzers by enumerating all available command line options, but this redesign alone is not sufficient. In order to clearly demonstrate this limitation, we introduce a naive multi-purpose fuzzer, which exhaustively enumerates all available command line options and thus would be a naive extension of current fuzzers.

A Naive Multi-Purpose Program Fuzzer. The key component of a naive multi-purpose program fuzzer is an operation input selector as shown in Figure 2. Using this selector, this fuzzer ensures to fuzz all available command line options for each mutated input file. In other words, for each mutated input file, it keeps executing a target program for all available command line options.

However, we found that such an enumeration over all command line options is inefficient. The problem is that if the mutated file input is invalid and thus early rejected by the parser, it is meant to be failed for all command line options. However, since this naive multi-purpose program fuzzer is not aware of such early rejection by the parser, it simply runs the target program with all command line options, all of which do not result in any benefit in terms of increasing the testing coverage.

We also observe that using a single command option shows limited execution coverage compared to using various command options. To clearly understand this, we have measured the edge coverage while providing a different number of command options (Figure 5). As shown in the figure, the edge coverage is increased as more command options are provide. In particular, comparing the edge coverage between a single command option and 5 command options, `ImageMagick`, `fig2dev`, and `MuPDF` show 53%, 154%, and 30% more edge coverage, respectively.

Our Approach. In order to efficiently fuzz multi-purpose programs, our key idea behind CrFuzz is that we leverage a machine learning technique to notice if an execution is terminated due to the early rejection by the parser or not. Specifically, CrFuzz aims at distinguishing following two cases: (i) when the execution is early terminated by the parser (i.e., the provided file input to the target program is invalid); and (ii) when the execution is proceeded to one of backend engines (i.e., the provided file input is valid). To this end, CrFuzz captures statistical differences in terms of execution coverage because aforementioned two cases must have different patterns in terms of execution coverage—i.e., a invalid file input case should not cover any of backend engines while a valid file input case should cover one of backend engines.

3 DESIGN

Now we describe the design of CrFuzz, an efficient multi-purpose program fuzzer (Figure 6). The core component of CrFuzz is its validity checker (§3.1), which performs a clustering analysis to determine if a given input is valid or not. CrFuzz first converts an execution feedback of an input into validity pair, a custom defined metric to efficiently measure input validity, which is then used for training the validity model.

CrFuzz leverages this validity checker while performing the fuzzing both file inputs and command line options (§3.2). Instead of enumerating all available command line options, CrFuzz first checks if a provided input file is invalid (through the validity checker) by executing the input file with one command line option. Then CrFuzz only attempts to enumerate other command line options only if the input file is valid.

3.1 Learning and Testing Validity

The goal of the validity checker is to check if a given execution feedback of an input is valid (i.e., the input is accepted by the parser of a target program) or invalid (i.e., the input is rejected by the parser). In order to check the validity, we leverage data mining or machine learning approaches. More specifically, we employ a

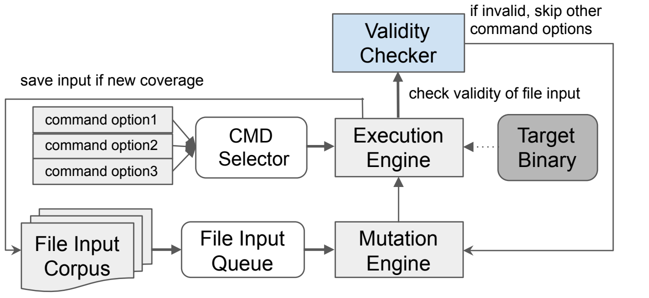


Figure 6: The overall workflow of CrFuzz, an efficient multi-purpose program fuzzer

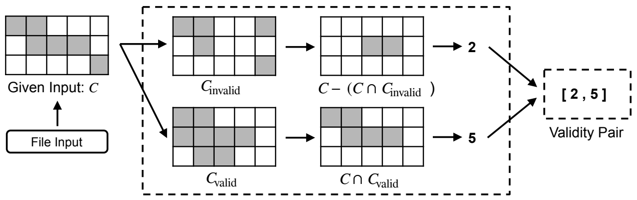


Figure 7: An example snapshot of showing how the validity pair is computed

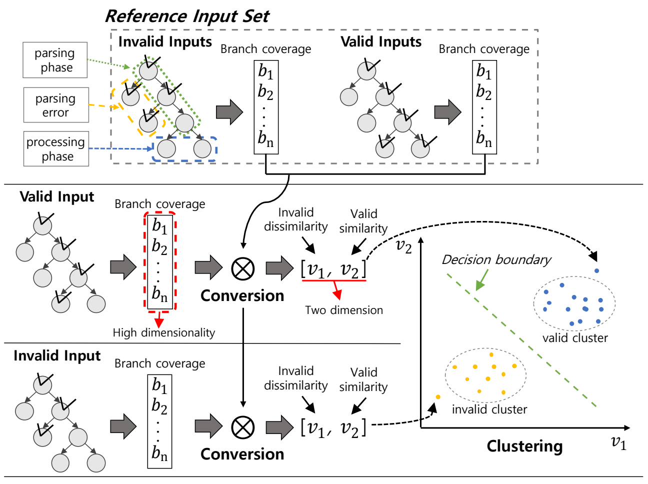


Figure 8: Overview of Validity Checker

cluster analysis (i.e., a unsupervised learning) such that each input can be grouped into either valid or invalid.

It is worth noting that supervised learning is not a suitable approach for the validity checking. This is largely due to the fact that it is difficult to obtain a large number of labeled data (i.e., a set of inputs labeled as valid or invalid). To obtain a labeled data, one would need perform a sophisticated program analysis on the parser of a target program. However, this would be challenging since the validity checking logic is hard-coded within the parser and understanding such a hard-coded logic requires non-trivial reverse-engineering efforts.

3.1.1 Validity Feature Selection. **Coverage as a Cluster Feature.** In performing statistical data analysis, one important thing is to select a good feature vector for the analysis. Particularly focusing on the clustering problem that CrFuzz attempts to solve, a desired feature vector would clearly distinguish two clusters, valid and invalid (i.e., maximizing the distance between valid and invalid clusters).

In this regard, coverage information of a corresponding input (which is provided as an execution feedback) may seem to be a good feature selection for CrFuzz. First, the coverage information is easy to obtain, as most fuzzers today are coverage-guided fuzzers which always produce the coverage information per execution. Second, the coverage information would have clear differences depending on whether target program's parser accepts (i.e., valid) or rejects (i.e., invalid). This is because accept/reject logic in the parser should follow a different code path, so the input should result in clearly different coverage information if accepted/rejected.

Limitation: Coverage as a Cluster Feature. However, we found that using a coverage information as it is has critical limitation, particularly with respect to clustering performance. This is due to the fact that using a coverage as a feature vector results in an extremely high dimension space, a well-known challenge in clustering. In other words, if the input feature has high dimension, the time complexity for clustering increases exponentially.

Specifically, if using the coverage, the number of dimension would be either the number of basic blocks (in the case of code-coverage guided fuzzers) or the number of edges (in the case of path-coverage guided fuzzers), and typical programs have very high number of basic blocks and edges. For instance, `FFmpeg` has 50k and 130k different basic blocks and edges, which is far beyond the number of dimensions that CrFuzz can handle.

Our Solution: Validity Pair as a Cluster Feature. CrFuzz defines a new metric, a validity pair, in order to overcome the limitation of simply using the coverage. The insight behind the validity pair is in using two reference coverages (which well represents valid/invalid input's coverage, respectively), thereby identifying important dimensions for validity clustering.

More precisely, the validity pair is defined as two scalar values (v_1, v_2), where

$$v_1 = C - (C \cap C_{\text{invalid}}) \text{ and}$$
$$v_2 = C \cap C_{\text{valid}}.$$

Here, C represents the coverage of a given input, and C_{valid} and C_{invalid} represent the reference coverage of a valid input and an invalid input, respectively. In other words, v_1 is designed to capture the coverage dissimilarity between the given input and the reference invalid input; and v_2 is to capture the coverage similarity between the given input and the reference valid input (the example is described in Figure 7).

This validity pair is a performance efficient metric for CrFuzz. Compared to naively using the coverage metric as it is, the validity pair only has two dimensions. Furthermore, it still captures the valid/invalid characteristics of a given input. This is because if two scalar values of a validity pair is higher, it implicates that the given input is more likely to be similar to a valid input.

3.1.2 Learning Validity Cluster. CrFuzz learns the validity clustering model using the aforementioned validity pair as the base metric. Specifically, CrFuzz performs k-means clustering, where k is two to represent valid and invalid clusters, in following steps (each step is described in Figure 8).

Preparing Reference Inputs. CrFuzz prepares the reference coverage of a valid input (i.e., C_{valid}) by manually collecting a reference valid input and then obtaining the execution coverage of it. The reference valid input in CrFuzz implies a well-known good input that is accepted by target program's parser. Most programs provide a sample working input, which demonstrates its basic features (i.e., a sample video/audio file for `FFmpeg` and a sample PDF file for `MuPDF`), and CrFuzz leverages such a sample working input. We note that it is not necessary for CrFuzz to have these sample inputs completely represent a valid behavior of the target program. Instead, it is sufficient for CrFuzz to have sample inputs not to be early rejected by the parsing logic, such that CrFuzz can distinguish whether a certain input would be early rejected or not.

In addition, CrFuzz prepares a reference invalid input by generating a completely random input—i.e., the value of each input byte as well as the size of the input is randomly determined. Since this input is randomly generated, it is very unlikely to be accepted by the target program's parser. As such, its execution coverage (i.e., C_{invalid}) would be well representing target program's early rejection behavior.

Clustering using Validity Pairs. For each new input (i.e., an input that is either generated or mutated for fuzzing), CrFuzz computes a validity pair (i.e., v_1 and v_2) based on the reference coverage of valid and invalid inputs (i.e., C_{valid} and C_{invalid}). Then the validity pair of each input is projected into a point in two dimensional euclidean space so as to compare euclidean distance between inputs. In other words, x-axis is represented with v_1 (i.e., the coverage dissimilarity from the reference invalid input) and y-axis is represented with v_2 (i.e., the coverage similarity from the reference valid input).

After that, CrFuzz performs a generic k-means clustering method. First, CrFuzz randomly selects two points at random (where each point represents each input) as initial centroids of two clusters. Then all other points are assigned to their closest centroid, where its distance measure is using the squared point-to-point euclidean distance (step 1). Then it re-computes two centroids by computing the average of all of each cluster's points (step 2). After updating all centroids, it re-assigns all points to their closest updated centroid, and then re-updates each centroids again. The aforementioned step 1 and step 2 are repeated until meeting one of two termination conditions. The first termination condition is that the sum of square euclidean distances reaches to the minimum value. More specifically, the objective function to be minimized is defined as L, where x is a data point, S_0 and S_1 are invalid and valid clusters, respectively, and c_i is centroid of S_i.

$$L = \underset{\mathbf{S}}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x_j \in S_i} \left\| x_j - c_i \right\|^2$$

CrFuzz computes sum of the squared euclidean distances between point and centroid of its cluster. Then CrFuzz repeats aforementioned process until this object function is minimized. The second termination condition is that the number of iteration reaches to the pre-defined maximum number. In our experiment, we used 100 as the maximum number.

Once the k-means clustering analysis is terminated, CrFuzz obtains two centroids. The centroid on the right-top side represents the centroid of a valid cluster, and the other centroid on the left-bottom side represents the centroid of an invalid cluster. This is because as an input point is more close to the right-top side, it is more likely indicating following two things: i) it is more dissimilar from the reference invalid input (i.e., v_1 is bigger than others) and ii) it is more similar to the reference valid input (i.e., v_2 is bigger than others).

Incremental Clustering. CrFuzz supports an incremental clustering in consideration of the inherent characteristic of fuzzing procedure. More specifically, fuzzers keep generating new inputs for testing, and such a new input can be kept being provided to CrFuzz, which can improve the accuracy of CrFuzz's cluster analysis. However, a downside of such an incremental clustering is runtime performance. If CrFuzz keeps computing mean square error (until it reaches the stable centroids) every time a new input is generated, it would significantly slow down the runtime performance of CrFuzz.

Therefore, CrFuzz attempts to strike a balance between cluster analysis accuracy and its runtime performances through dynamically adjusting the frequency of an incremental clustering analysis. The key insight behind this is that inputs provided during the initial phase are impacting the accuracy far more than the inputs provided during the later phase. Thus, CrFuzz gradually decreases the frequency of the incremental analysis—i.e., decreasing the frequency as a factor of two, whenever the number of provided inputs are doubled. For instance, If the number of provided inputs is less than 2^8, CrFuzz performs incremental clustering every time a new input arrives. If the number of provided inputs is between 2^8 and 2^9, CrFuzz performs incremental clustering when every two new inputs arrive.

3.1.3 Testing Validity. CrFuzz utilizes aforementioned a validity model to check if a new input is valid or invalid. To be specific, it first computes a validity pair of a new input, and computes euclidean distances from valid and invalid centroids, respectively. Then it is determined to be either valid or invalid if it is more close to valid or invalid centroids, respectively.

3.2 Multi-purpose Program Fuzzing

CrFuzz utilizes validity checker to address the limitation of naive multi-purpose program fuzzers. In particular, it executes all command options only if the provided file input is predicted to be a valid file input. If not, CrFuzz stops fuzzing other command options, which allows CrFuzz to significantly save the fuzzing time.

To better illustrate this, Figure 4 depicts the high level comparison between naive multi-purpose program fuzzing and CrFuzz's multi-purpose program fuzzing, particularly highlighting how CrFuzz can efficiently handle various command options. In this figure, each file input is fuzzed with all different command options (represented as ① in y-axis). After enumerating all command options, the fuzzer generates (or mutates) another file input is fuzzed (represented as ② in x-axis). In the case of the naive multi-purpose program fuzzer, it would execute all combinations (i.e., all boxes in the figure) as it cannot notice if a provided input is valid or not. On the contrary, CrFuzz's multi-purpose program fuzzer can predict if a provided input is valid or not (using the validity checker), so it does not need to fuzz other command options if the provided input file is predicted to be invalid. As a result, CrFuzz's fuzzer only needs to execute the gray boxes, which is significantly less than all the boxes that the naive fuzzer should fuzz.

Algorithm 1 Algorithm of CrFuzz's multi-purpose program fuzzing loop

```
$Q_f = Q_c = C_{\text{invalid}} = C_{\text{valid}} = 0$;
load $Q_f$ and $Q_c$ with the provided valid file input(s) and cmd options respectively;
compute $C_{\text{invalid}}$ and $C_{\text{valid}}$;
while true do
    $f$ = DEQUEUE($Q_f$);
    $f'$ = MUTATE($f$);
    for $c$ in $Q_c$ do
        execute program with $f'$ and $c$;
        compute $(v_1, v_2)$ of $f'$;
        if find new coverage then
            $Q_f$ = ENQUEUE($f'$);
            if find $n$ new inputs then
                perform incremental clustering;
        if $c$ is first cmd option then
            VALIDITYCHECKER($v_1, v_2$);
            if $f'$ is invalid then
                break;
```

More specifically, we designed CrFuzz to skip over unnecessary command options if the provided input is predicted to be invalid (the detailed algorithm is described in Algorithm 1).

4 IMPLEMENTATION

We implemented CrFuzz, and applied it for three different state-of-the-art fuzzers, AFL, QSYM, and MOpt. Since all these fuzzers are not designed to support multi-purpose program fuzzing, we first implemented a naive multi-purpose program fuzzer of these as follows. In the case of AFL and MOpt, we first modified the fork server of AFL and MOpt as its fork server routine is hardcoded for a single command line option. Hence, we extended such a routine to support multiple command line options. Then for each mutated file input that AFL and MOpt are generating, we ensured that all command line options are executed in order. In the case of QSYM, we replaced two original AFL instances of QSYM into two aforementioned native multi-purpose AFL fuzzer instances—the original QSYM runs three instances in parallel, two AFL instances and a single concolic execution instance.

We applied CrFuzz to these naive multi-purpose program fuzzers, which we call AFL+CrFuzz for AFL, QSYM+CrFuzz for QSYM, and MOpt+CrFuzz for MOpt, respectively. To be more specific, for each fuzzer we first implemented the validity checker as described in §3.1, which converts the coverage feedback into the validity pair and performs the k-means clustering. Then we first run the mutated file with one command line option, then check the input validity of an execution through the validity checker. If predicted to be an invalid input, CrFuzz augmented fuzzers stop fuzzing other command line options but proceeding to fuzz another mutated file.

Table 1: Fuzzer settings for evaluation

Fuzzers	***Setup***	
	Instances	Description
AFL [4]	1 AFL master and 2 AFL slaves	-
AFL+CrFuzz	1 AFL+CrFuzz master and 2 AFL+CrFuzz slaves	-
QSYM [3]	1 concolic executor; 1 AFL master and slave each	using docker ([3])
QSYM+CrFuzz	1 concolic executor; 1 AFL+CrFuzz master and slave each	using docker ([3])
MOpt [1]	1 MOpt master and 2 MOpt slaves	option "-L 0"
MOpt+CrFuzz	1 MOpt+CrFuzz master and 2 MOpt+CrFuzz slaves	option "-L 0"

5 EVALUATION

Experimental Setup. In this evaluation, we performed an experiment with CrFuzz augmented fuzzers, AFL+CrFuzz, QSYM+CrFuzz, and MOpt+CrFuzz, where its fuzzing instance configuration is listed in Table 1. Under this configuration, we ran nine multi-purpose programs (`Graphicsmagick`, `MuPDF`, `Xpdf`, `ImageMagick`, `Xfig`, `libsixel`, `libtiff`, `FFmpeg`, `GhostScript`) for 48 CPU hours, and then repeated the experiment for five times. All our experiments were carried out a machine of Intel Xeon Gold 6140 with 32 CPU cores and 512GB RAM, which runs Ubuntu 18.04 LTS.

Research Questions. In the following of this evaluation section, we aim to answer following research questions:

RQ1. What is the accuracy of CrFuzz's clustering analysis in its validity checker?

RQ2. Does CrFuzz's fuzzing approach truly improve the multi-purpose program fuzzing capability, particularly with respect to path and edge coverage?

RQ3. What are the new vulnerabilities that are discovered by CrFuzz?

5.1 Accuracy of Validity Checker (RQ1)

As described in §3.1, CrFuzz leverages validity checking capability for fuzzing. Since the validity checker performs the clustering analysis, CrFuzz's overall fuzzing performance relies on the accuracy of the clustering analysis. As such, this subsection analyzes such accuracy of CrFuzz's validity checker. In order to collect the ground truth (i.e., a set of files labeled as either a valid or invalid file), we manually analyzed the source code of following three target programs, including `FFmpeg`, `libtiff`, and `Ghostscript`. Then we identified the termination routine due to both parsing issues and non-parsing issues. We instrumented such termination routine such that we can mark a label suggesting which input file is valid or not.

Given these labeled files, we first plotted all data points (Figure 9) for each program. When plotting, we projected each data point according to the validity pair distance metric as described in §3.1.1. In these figures, each red cross mark denotes an invalid input, and each blue circle denotes a valid file. It can be observed that overall red crosses are located closed to the bottom-left corner while blue circles are located closed to the top-right corner. This overall location pattern implies that the validity pair is an effective metric capturing the execution differences between valid and invalid files.

Table 2: Accuracy of Validity Checker

Program	*Description*	*Version*	*Actual (total exec)*		*Inferred (total exec)*				r_{FP}	r_{FN}
			Invalid	Valid	Invalid	FN	Valid	FP		
FFmpeg	mp4 to mov	4.1	1.36M	203K	1.36M	3.96K	203K	3.10K	0.23%	1.95%
libtiff	tiff to ps	4.1.0	64.3M	6.82M	63.7M	70.4K	7.36M	610K	0.95%	1.03%
GhostScript	ps to pgm	9.51rc1	181K	66.9K	181K	27	67.1K	137	0.76%	0.20%

We acknowledge that the each plot shows some overlaps (i.e., it may not seem straight-forward to draw a decision boundary bisecting two labels), but we would like to emphasize the number of those overlapped data points are relatively small compared to the number of entire data points, which we further evaluate next.

In order to clearly evaluate the clustering accuracy, we measured the false positive and false negatives of CrFuzz's k-means clustering. More precisely, we measured false positives (r_{FP}) and false negatives (r_{FN}) as follows.

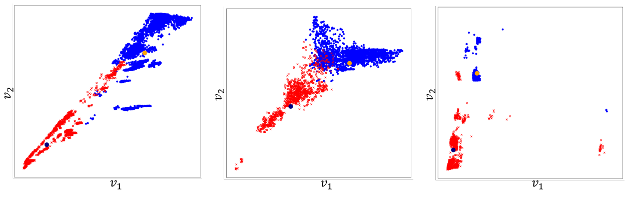
$$r_{\mathrm{FP}} = N_{\mathrm{FP}}/N_i * 100 \text{ and}$$
$$r_{\mathrm{FN}} = N_{\mathrm{FN}}/N_v * 100.$$

In these two measures, N_{FP} and N_{FN} denote the number of false positive and false negative in total executions. N_i and N_v denote the total number of executions with invalid and valid files, respectively. The result of false positives and false negatives are shown in Table 2. Overall r_{FP} is always less than 1%. It is worth noting that if the false positive occurs (i.e., an invalid file is predicted to be valid), CrFuzz would not leverage the performance optimization of CrFuzz as it would enumerate all command line options without much coverage benefits. On the other hand, r_{FN} is measured in the range from 0.2% to 2%. If the false negative happens (i.e., a valid file is predicted to be invalid), CrFuzz would stop trying more command line options although it would be more likely beneficial to try more options with respect to extend the execution coverage. Given these results on r_{FP} and r_{FN}, although CrFuzz's clustering analysis does not yield perfect accuracy, we believe its result is good enough to significantly augment the fuzzing capability for multi-purpose programs, which we demonstrate in the next subsection.

Runtime Speed of Clustering Analysis. Over the entire 48 hours of fuzzing time, the validity checker used eight minutes on average (i.e., three minutes for clustering and five minutes for predicting the validity), which only account for 0.28% of entire fuzzing time. Based on this result, we believe that the dimension reduction with the validity pair (along with incremental learning features) is efficient enough to fuzz multi-purpose programs.

5.2 Fuzzing Efficiency of CrFuzz (RQ2)

Naive Vs. CrFuzz-Augmented Frruzzers. Compared to a naive multi-purpose program fuzzer, CrFuzz leverages the feedback from the validity checker to accelerate the fuzzing performances. In particular, we implemented both naive and CrFuzz versions for three fuzzers, AFL, QSYM, and MOpt (described in §4). Using these fuzzers, we ran nine multi-purpose programs for 48 CPU hours (repeated five times). Then we compared its average performance with respect to path coverage and edge coverage as shown in Table 3 and Table 4, respectively. According to the result, AFL+CrFuzz, QSYM+CrFuzz and MOpt+CrFuzz achieved better path and edge



(a) FFmpeg (b) libtiff (c) GhostScript

Figure 9: Plotted results of input files according to the validity pair distance metric (i.e., x-axis represents v_1 and y-axis represents v_2). Each red cross represents an invalid file and each blue dot represents a valid file.

coverage for all nine multi-purpose programs than the naive version. On average, CrFuzz showed 19.3% better path coverage and 5.68% better edge coverage to that of the naive multi-purpose fuzzer, demonstrating CrFuzz's efficiency in exploring more testing coverage for multi-purpose programs.

Efficiency While Varying Command Line Options. In order to understand the fuzzing efficiency over the number of command line options, we varied the number of command line options and compared the fuzzing performance between the naive and CrFuzz-augmented versions (Figure 10). For both version of fuzzers, as more command line options are provided, its path and edge coverage are extended as it enables to explore more features in the backend engines. However, it is noticeable that CrFuzz augmented versions are always show better performance, and such performance advantage gap is increased more as more command line options are given (i.e., the gap between three and six command options). This is because as more command line options are given, the search space for command line options are doubled. Hence, naive multi-purpose fuzzers would waste more of its fuzzing time on invalid files with more command line options, while CrFuzz-augmented multi-purpose fuzzers can save its fuzzing time by quickly dropping such invalid files with the help from the validity checker.

5.3 New Vulnerabilities Found by CrFuzz (RQ3)

Over the course of evaluating CrFuzz, we kept running CrFuzz-augmented fuzzers for two weeks against nine multi-purpose programs to find new vulnerabilities. To summarize, CrFuzz found 277 new vulnerabilities in total, where 212 of those are already confirmed and accordingly fixed by the respective vendors as shown in Table 5. We highlight that many target programs (particularly `Graphicsmagick`, `MuPDF`, `ImageMagick`, `libtiff`, `FFmpeg`, and `GhostScript`) are fuzzed by many security researchers due to its popularity and

Table 3: Path coverage found in multi-purpose programs using three command line options w/o and w/ CrFuzz in 48 CPU hours

Program	***Path Coverage***					
	AFL		QSYM		MOpt	
	Orig.	+CrFuzz	Orig.	+CrFuzz	Orig.	+CrFuzz
FFmpeg	11,321	16,403 (+44.89%)	12,467	17,090 (+37.08%)	15,420	17,379 (+12.70%)
Graphicsmagick	5,938	7,263 +(22.31%)	4,256	5,703 (+34.01%)	6,644	7,130 (+7.31%)
MuPDF	2,069	2,147 (+3.74%)	2,124	2,241 (+5.53%)	2,426	2,448 (+0.91%)
Xpdf	2,957	3,408 (+15.23%)	2,376	2,502 (+5.32%)	2,308	4,015 (+73.96%)
ImageMagick	5,938	7,263 (+22.31%)	4,242	5,975 (+40.84%)	5,450	6,238 (+14.46%)
Xfig	4,261	4,787 (+12.33%)	3,208	3,537 (+10.27%)	4,158	5,390 (+29.63%)
libsixel	4,070	4,724 (+16.01%)	4,368	4,466 (+2.24%)	4,101	4,313 (+5.19%)
libtiff	5,071	5,350 (+5.50%)	4,183	5,036 (+20.38%)	4,924	6,228 (+26.48%)
GhostScript	2,591	2,962 (+14.32%)	-	-	2,564	3,150 (+22.85%)
Average	+17.40%		+19.46%		+21.50%	

Table 4: Edge coverage found in multi-purpose programs using three command line options w/o and w/ CrFuzz in 48 CPU hours

Program	***Edge Coverage***					
	AFL		QSYM		MOpt	
	Orig.	+CrFuzz	Orig.	+CrFuzz	Orig.	+CrFuzz
FFmpeg	41,652	46,685 (+12.08%)	41,866	47,759 (+14.08%)	46,620	49,949 (+7.14%)
Graphicsmagick	9,933	10,141 (+2.09%)	9,090	9,530 (+4.84%)	10,244	10,320 (+0.74%)
MuPDF	4,165	4,197 (+0.78%)	4,195	4,204 (+0.20%)	4,239	4,285 (+1.09%)
Xpdf	6,172	6,344 (+2.78%)	5,832	5,912 (+1.38%)	5,830	8,241 (+41.36%)
ImageMagick	11,971	13,454 (+12.39%)	11,715	12,128 (+3.53%)	11,024	12,380 (+12.30%)
Xfig	4,770	4,774 (+0.08%)	4,493	4,642 (+3.32%)	4,734	4,875 (+2.98%)
libsixel	5,043	5,271 (+4.51%)	5,452	5,578 (+2.31%)	5,041	5,257 (+ 4.28%)
libtiff	7,686	7,730 (+0.57%)	7,544	8,014 (+6.23%)	7,754	8,122 (+4.74%)
GhostScript	28,590	28,891 (+1.05%)	-	-	28,891	29,123 (+2.10%)
Average	+4.04%		+4.49%		+8.53%	

thus those are fuzzed for numerous CPU hours by far. More importantly, all of those (except GhostScript) are selected by Google's OSS-Fuzz project [2] and heavily fuzzed for last three years. However, in spite of such heavy fuzzing efforts towards these programs CrFuzz discovered a striking number of new vulnerabilities, demonstrating significantly better fuzzing capability of CrFuzz.

In the case of Graphicsmagick, it takes bmp image files as input and convert it to other formatted files (such as vector image files like pdf), which identified 3 new vulnerabilities. For mutool, we found seven vulnerabilities when it resizes the pdf images. ImageMagick provides over 100 command line options, and CrFuzz identified 42 vulnerabilities. Xfig takes fig files as input and converts it to 40 different formats. CrFuzz identified 14 vulnerabilities, all of which found in backend engines. libsixel converts png files into sixel files, and we identified multiple bugs from the command which converts image colors. libtiff takes tiff files as input and convert it to various levels of ps files (the default configuration is set to be level 1), and CrFuzz identified the vulnerability which is only activated when the level is 3. FFmpeg is a popular video image decoder, which provides over 200 different command line options. CrFuzz identified 153 different vulnerabilities, most of which are identified from its backend engine.

6 RELATED WORK

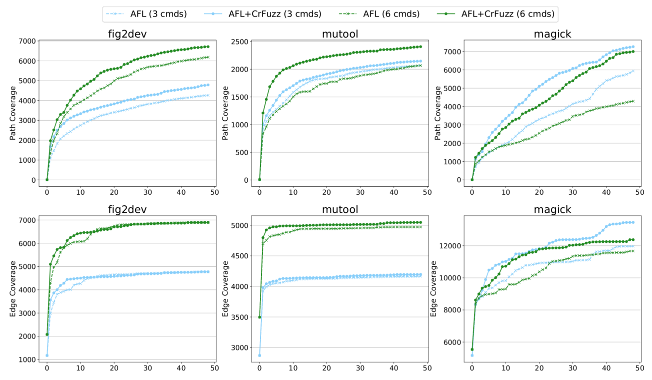
Advanced Fuzz Scheduling. AFLfast [10] and fair-fuzz [20] change seed scheduling algorithm to prioritize rarely-exercised branches for higher coverage. MOpt [25] uses a customized Particle Swarm Optimization (PSO) algorithm to optimize the mutation operation scheduler. Cerebro [23] utilizes various factors such as code complexity, coverage, and execution time to improve seed scheduling strategy. AFLGO [11] and Hawkeye [12] leverage seed scheduling to guide the fuzzer towards target locations of program based on distance metrics.

Solving Hard Constraints. To overcome the limitation of solving hard constraints such as magic bytes, various approaches have been suggested. For example, Steelix [22] and Laf-intel [5] split magic bytes to make them as weak constraint with extra implemetation. Vuzzer [30], Angora [13] and Matryoshka [14] use taint-analysis to help the fuzzer solve constraints through control- and data-flow information. REDQUEEN [8] utilizes program-state analysis to solve magic bytes and checksum without taint or symbolic execution.

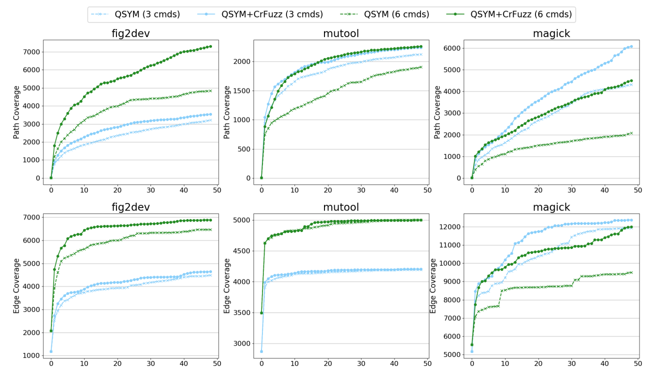
Table 5: Newly Discovered Vulnerabilities by CrFuzz

Program	Version	Vulnerability Type	Found	Fixed
FFmpeg	4.1 4.2	stack-buffer-overflow	1	1
		heap-buffer-overflow	23	18
		heap-use-after-free	4	0
		memory leak	23	19
		assertion	1	1
		invalid free	1	1
		segmentation fault	8	6
		division by zero	19	3
		signed integer overflow	18	9
		outside the range of representable	6	2
		out-of-bound	2	2
		left shift of negative value	33	29
		left shift cannot be represented in type `int`	7	6
		null pointer passed as argument	5	4
		load of null pointer	1	1
		pointer index expression overflowed	1	1
GhostScript	9.51rc1	global-buffer-overflow	6	6
		stack-buffer-overflow	3	2
		heap-buffer-overflow	24	21
		heap-use-after-free	3	2
		memory leak	1	1
		segmentation fault	7	5
		division by zero	5	4
libtiff	4.1.0	heap-buffer-overflow	1	0
libsixel	1.8.3 1.8.4	heap-buffer-overflow	6	6
		memory leak	2	2
Xfig	3.2.7b	global-buffer-overflow	6	6
		stack-buffer-overflow	3	3
		heap-buffer-overflow	2	2
		segmentation fault	3	3
MuPDF	1.15.0	heap-buffer-overflow	3	1
		heap-use-after-free	2	0
		assertion	2	0
ImageMagick	7.0.8-68 7.0.9-0	heap-buffer-overflow	6	6
		heap-use-after-free	2	2
		memory leak	4	4
		division by zero	6	6
		signed integer overflow	5	5
		outside the range of representable	16	16
		shift exponent is too large	1	1
		unsigned offset overflowed	2	2
Graphicsmagick	1.3.34	heap-buffer-overflow	2	2
		assertion	1	1
Total			**277**	**212**

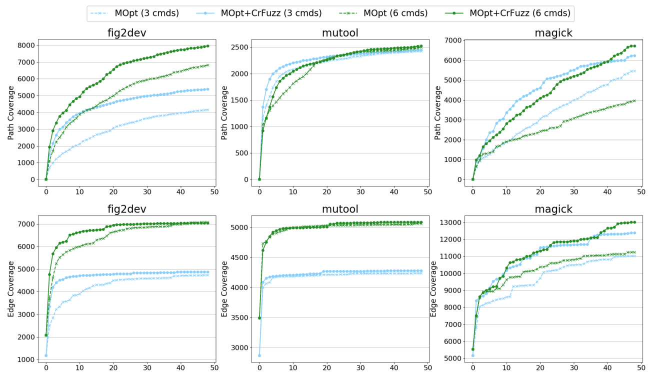
T-Fuzz [28] removes sanitiy checks in the programs to bypass the hard constraints. Hybrid fuzzers [15, 16, 32, 35, 39, 40] leverage a concolic executor to solve the hard constraints. For example, QSYM [39] and Intriguer [16] optimize symbolic emulation for a fast concolic executor. Savior [15] guides the concolic executor towards the bug in the program.



(a) AFL



(b) QSYM



(c) MOpt

Figure 10: Path and Edge Coverage test using three and six command-line options. The x-axis reprents the hour, and y-axis represent the coverage: the y-axis of top graphs represents the path coverage, and y-axis of bottom graphs represents the edge coverage.

Input-Aware Fuzzing. Profuzzer [38] infers the input type of each byte to enhance the mutation efficiency by probing changes of program behaviors (edge coverage change). By doing so, it improves the mutation strategy to generate semantically-valid input to find deep bugs. Recent studies [7, 9, 27, 34] guide the fuzzer to generate highly-structured inputs by using coverage-feedback. For example, Nautilus leverages grammar specification to better generate and mutate the test inputs with coverage guidance. In constrast, Grimoire leverages coverage-feedback to synthesize *highly-structured* inputs without any form of human interaction.

Improving Fitness Function. CollAFL [17] and InsTrim [19] enhance the way of coverage instrumentation to achieve more accurate and lightweight coverage information. TortoiseFuzz [36] and Ankou [26] propose new coverage evaluation techinques for better seed scheduling. Some studies [21, 29, 37] have focused on detecting algorithmic complexity vulnerabilities based on new coverage metrics such as resource usage or execution path length.

Learning-based Fuzzing. Recent studies use learning techniques to help the fuzzers to increase the coverage and find bugs. Learn&fuzz [18] uses RNN (Recurrent Neural Network) to generate valid highly-structured inputs. Skyfire [33] learns a probabilistic contextsensitive grammar (PCSG) from highly-structured inputs to generate well-distributed seeds. Angora [13] uses taint analysis and gradient descent to solve the hard constraints. In this approach, the taint analysis tells the position of magic bytes to the fuzzer, and then gradient descent shows how to mutate that position of bytes to solve the constraint. NEUZZ [31] uses deep-learning to synthesize the program behaviors to improve the coverage. FuzzGuard [41], which is used in directed-guided fuzzing, uses deep-learning to filter out unreachable input without executing it.

7 CONCLUSION

This paper proposed CrFuzz, an efficient multi-purpose program fuzzer. It implements a clustering analysis to predict if a new input file would be accepted by a target program or not. Utilizing this clustering analysis, it redesigns current state-of-the-art-fuzzers, including AFL, QSYM, and MOpt to efficiently fuzz multi-purpose program. According to the evaluation, CrFuzz-augmented fuzzers have shown reasonable better coverage as well as uncovering 277 new vulnerabilities in various software programs.

8 ACKNOWLEDGMENT

We thank anonymous reviewers for their insightful comments, which significantly improved the final version of this paper. This work was partially supported by Agency for Defense Development (ADD) in South Korea (No. UE191003ED) and by the National Research Foundation (NRF) of South Korea grant funded by the Korean government (MSIT) (No. 2019R1C1C1006095).

REFERENCES

[1] Mopt source code. https://github.com/puppet-meteor/MOpt-AFL.
[2] Oss-fuzz - continuous fuzzing of open source software. https://github.com/google/oss-fuzz.
[3] Qsym source code. https://github.com/sslab-gatech/qsym.
[4] American fuzzy lop. http://lcamtuf.coredump.cx/afl/.
[5] Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/.
[6] syzkaller is an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.
[7] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Proceedings of the 2019 Annual*

Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2019.

[8] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[9] T. Blazytko, M. Bishop, C. Aschermann, J. Cappos, M. Schlögel, N. Korshun, A. Abbasi, M. Schweighauser, S. Schinzel, S. Schumilo, et al. {GRIMOIRE}: Synthesizing structure while fuzzing. In *Proceedings of the 28th USENIX Security Symposium (Security)*, SANTA CLARA, CA, Aug. 2019.

[10] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[11] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[12] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[13] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, SAN FRANCISCO, CA, May 2018.

[14] P. Chen, J. Liu, and H. Chen. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[15] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. Savior: Towards bug-driven hybrid testing. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, SAN FRANCISCO, CA, May 2020.

[16] M. Cho, S. Kim, and T. Kwon. Intriguer: Field-level constraint solving for hybrid fuzzing. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[17] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, SAN FRANCISCO, CA, May 2018.

[18] P. Godefroid, H. Peleg, and R. Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana-Champaign IL, Sept. 2017.

[19] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao, and S.-K. Huang. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[20] C. Lemieux and K. Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Montpellier, France, Sept. 2018.

[21] C. Lemieux, R. Padhye, K. Sen, and D. Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th International Symposium on Software Testing and Analysis (ISSTA)*, Amsterdam, The Netherlands, July 2018.

[22] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 25th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Paderborn, Germany, Sept. 2017.

[23] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 24th European Software Engineering Conference (ESEC) / 27st ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Tallinn, Estonia, Aug. 2019.

[24] libfuzzer. libfuzzer. https://llvm.org/docs/LibFuzzer.html.

[25] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Security Symposium (Security)*, SANTA CLARA, CA, Aug. 2019.

[26] V. J. Manès, S. Kim, and S. K. Cha. Ankou: Guiding grey-box fuzzing towards combinatorial difference. May 2020.

[27] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, July 2014.

[28] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: fuzzing by program transformation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, SAN FRANCISCO, CA, May 2018.

[29] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[30] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[31] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, SAN FRANCISCO, CA, May 2019.

[32] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[33] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[34] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, QC, Canada, May 2019.

[35] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun. Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, Gothenburg, Sweden, May 2018.

[36] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. May 2020.

[37] J. Wei, J. Chen, Y. Feng, K. Ferles, and I. Dillig. Singularity: Pattern fuzzing for worst case complexity. In *Proceedings of the 23th European Software Engineering Conference (ESEC) / 26st ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Lake Buena Vista, Florida, Nov. 2018.

[38] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, SAN FRANCISCO, CA, May 2019.

[39] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*, BALTIMORE, MD, Aug. 2018.

[40] L. Zhao, Y. Duan, H. Yin, and J. Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[41] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. Aug. 2020.