Neurocomputing 460 (2021) 195–204

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

RapidFuzz: Accelerating fuzzing via Generative Adversarial Networks

Aoshuang Ye, Lina Wang *, Lei Zhao *, Jianpeng Ke, Wenqi Wang, Qinliang Liu

Key Laboratory of Aerospace Information Security and Trust Computing, Ministry of Education, China
School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China

ARTICLE INFO

Article history:
Received 12 January 2021
Revised 9 May 2021
Accepted 25 June 2021
Available online 29 June 2021
Communicated by Zidong Wang

Keywords:
AFL
Fuzzing
Generative Adversarial Networks

ABSTRACT

We implement a Generative Adversarial Network (GAN) based fuzzer called RapidFuzz to generate synthetic testcase, which can precisely catch the data structure feature in a relatively shorter time than the state-of-art fuzzers. RapidFuzz provides potential seeds generated by GAN. i.e., The generated seeds with similar but different numerical distributions accelerate the mutation process. An algorithm is elaborately designed to locate the hot-points generated by GAN. The generated testcases make structural features easier to be identified, which makes the whole process faster. In our experiment, RapidFuzz considerably improves the performance of American Fuzzy Lop(AFL) in speed, coverage, and mapsize. We select 9 open-sourced programs with different highly-structured inputs to demonstrate the effectiveness of RapidFuzz. As a result, code coverage is significantly improved. For tiff2pdf and tiffdump, coverage increase exceeds over 20%. We also observe that RapidFuzz achieves the same coverage with less time than AFL. Furthermore, AFL absorbs 21% of generated seed files in tiff2pdf with an average absorption rate around 15% in other programs.

© 2021 Elsevier B.V. All rights reserved.

1. Introduction

Evolutionary fuzzing has become one of the most popular vulnerability discovery solutions both on software and firmware [1,2], which is extensively used and studied in the security community. Coverage-guided fuzzing methods like AFL achieve significant success [3]. The state-of-art coverage-guided fuzzers, including libFuzzer, honggfuzz, AFL, and etc, have continuously contributed to discovering thousands of vulnerabilities. In general, fuzzing test aims to detect unintended program behaviors and discover bugs by generating and sending a large amount of test inputs to the target program.

With respect to the construction of testcases, fuzzing techniques can be divided into two categories: mutation-based and generation-based fuzzing techniques. Generation-based fuzzers aim to generate highly-structured testcases based on the grammars of program inputs. However, because the construction of specific grammars is mainly done manually and is proven to be laborious, the generation-based techniques seem inefficient. On the contrary, mutation-based fuzzers generate new testcases by mutating existing testcases, also known as initial seeds. Mutation-based methods require nearly no knowledge about the information input structures, which are more scalable to numerous applications.

Therefore, the quality of mutated testcases is one of the most important factors influencing the effectiveness and efficiency of mutation-based fuzzers. Typically, there is abundant structural information in program inputs. A program often checks the syntactic format of inputs and block abnormal inputs. As mutation-based fuzzing techniques generate testcases by almost randomly mutating existing ones, the space for mutation is considerably large, especially for highly-structured program inputs (e.g., PDF, PNG, TIFF). Consequently, for highly-structured data, it is inefficient to generate testcases that can pass their syntactic checking and explore the in-depth states in programs [4].

Among the existing mutation-based fuzzing tests, machine learning-based methods have attracted much attention. Learn&Fuzz utilizes RNN to learn grammar structure automatically. However, Learn&Fuzz does not put their concentration on the speed. Therefore, although RNN has made progress in coverage via learning grammar, the lack of speed makes the entire process inefficient. Furthermore, existing learning-based methods tend to collect their training dataset from the pre-prepared dataset. This process also limits the spread of its usage. In order to solve these problems, we propose an alternative method of data collection

* Corresponding authors at: School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China.

E-mail addresses: yasfrost@whu.edu.cn (A. Ye), lnwang@whu.edu.cn (L. Wang), leizhao@whu.edu.cn (L. Zhao), kejianpeng@whu.edu.cn (J. Ke), wangwenqi_001@whu.edu.cn (W. Wang), liuqinl@whu.edu.cn (Q. Liu).

https://doi.org/10.1016/j.neucom.2021.06.082
0925-2312/© 2021 Elsevier B.V. All rights reserved.

and apply a further advanced learning model to learn the specific structure more rapidly.

To address the challenges of generating highly-structured program inputs, researchers have proposed multiple learning-based techniques. Learn&Fuzz [5] trains a recurrent neural network (RNN) to obtain the grammatical structure. FasterFuzzing [6] uses the learning model just for data augmentation. FasterFuzzing can be seen as another augmentation strategy to make up for the deficiency of the quantity of input samples. GanFuzz [7] starts to show the power of GAN in generating highly-structured inputs but only focuses on industrial protocols.

These techniques have shown their effectiveness in generating highly-structured program inputs. These training models are constructed previously on a large amount of existing samples. For unknown program inputs such as proprietary protocols, they may lose the advantages. Moreover, how to explore the space of mutation or accelerate the speed of generating highly-structured inputs has not been investigated.

We propose RapidFuzz, a novel method with implementing advanced Wassertein-Generative Adversarial Networks with Gradient Penalty(WGAN-GP) as generation model. First, RapidFuzz addresses various data formats, which shows great augmentability of GAN based fuzzing to handle complex data. Second, different with the RNN based methods, RapidFuzz directly learns the numerical distribution of seed samples. Combining this feature with our sewing algorithm, RapidFuzz can accelerate the state-of-art fuzzer like AFL and FairFuzz and improve code coverage. In our experiments, we utilizes a series of t-SNE graphs to show the difference of the testcase distribution of RapidFuzz and AFL. After sufficient mutation by RapidFuzz, testcases remains clustering but with expanded distribution. Our contributions can be organized as follows:

- First GAN-based fuzzing method focuses on both coverage, speed, and highly-structured format.
- Combining with mutation gene detection algorithm and advanced learning model WGAN-GP, the distribution of generated testcase is expanded.
- RapidFuzz can deal with unknown highly-structured data without prepared training data.
- Experimenting on 9 programs with 6 highly-structured formats, which shows strong augmentability.

This paper is organized as following structure. Section II introduces related work respectively to detail the contribution. Section III shows the methodology in this paper. Section IV gives the evaluation metrics and discusses about experiment results. Section V gives a brief conclusion and specifies some future work.

2. Related work

Since fuzzing, symbolic execution [8,9], and dynamic taint analysis [10,11] have been proven to be practical automatic testing approaches for discovering software vulnerability. Fuzzing is regarded as one of the most efficient methods in the vulnerability detection domain [12]. Fuzzing is proposed to ensure the stability of UNIX programs in 1990 [13]. It uses random and unexpected inputs to reveal the potential crashes or bugs with massive number of trials. Fuzzing is usually regarded as security ensuring tool by multinational software companies like Google, Adobe, and Microsoft [14]. Fuzzing usually uses massive testcases to explore the state space of target program. This intuition is relatively simple but effective by using unexpected inputs. Nevertheless, entirely randomly generated testcases do not always achieve satisfying results. Previous attempts try to solve this problem from two aspects, which are input mutation and generation. The mutation-based fuzzing test modifies the existing testcases by applying mutation operations. Those testcases created by the modification can successfully verify systems with unstructured inputs because the programs can smoothly accept unstructured data without the syntax parsing. However, it is not difficult to understand that the mutation strategy determines both the speed and effect of the entire fuzzing process. For those highly-structured data (e.g., PDF, PNG, TIFF), mutated testcases find it challenging to pass the syntax parsing to explore the in-depth code. Although the generation-based fuzzing is regarded as a suitable tool for dealing with highly-structured data, it still lacks efficiency. Traditional generation-based fuzzing tests are mostly based on customed grammar. This process relies on manual work and is proven to be laborious.

The mutation-based fuzzing creates testcases from existing data by leveraging modification. The most basic mutation strategy is randomness [13]. AFL [15] uses a priority queue to collect program coverage for each testcase to guide the mutation process. It combines genetic algorithm with flipping, replacing, or other mutation operations to create testcases. BuzzFuzz [16] and TaintScope [17] utilizes dynamic taint tracing to automatically locate input regions that influence values at library calls [18]. The mutation strategy of AFLFast [19] is modeled with the Markov chain, which gives more probability to the testcases that trigger low-frequency paths. AFLGo [20] uses the control flow graphs to optimize seed mutation. VUzzer [21] utilizes the seed mutation strategy based on both data flow and control flow to extract program features via lightweight static analysis and dynamic analysis. FairFuzz [22] design a mechanism to discover the low-frequency path in AFL. EcoFuzz leverages reward probability based system to trigger potential path [23]. All of these methods, to achieve the perfect software test, aims at exploring program states as many as they can. The main challenge is that computation ability and time are relatively limited compared to the explosive exploring space. i.e., Slightly guided mutation or none guided fuzzing methods can not deal with the specific structure (e.g., checksum) well.

Recently, the generation-based fuzzing that uses deep learning method is beginning to emerge rapidly. Learn&Fuzz [5] trains a recurrent neural network (RNN) to obtain the grammatical structure of a pdf.obj file. Comparing with the conventional grammar-based model, this idea deserves some merits in code coverage. RNN is effective and widely used in the text processing area. Therefore, utilizing RNN as a generator for grammar seems reasonable. Although RNN has made progress in coverage, it does not seem to have many advantages in speed. In response to this problem, FasterFuzzing firstly proposes replacing RNN with GAN. GAN directly generates new samples by learning the statistical distribution of collected samples. The testcases generated by GAN resemble the authentic testcases but remain slightly different. This slight difference obeys the distribution of the collected data. FasterFuzzing [6] shows its experiment on a small scale program in cpp-ethereum, which achieves certain results compared to the RNN model. Though the GAN-based model is implemented in FasterFuzzing, the scale of testcases used in the experiment are still too minor to make sense. The reliable experiment data to prove its validity is still insufficient. Moreover, Ganfuzz proposes a framework that uses industrial protocols as the primary test inputs. Ganfuzz implements a model that utilizes RNN as the generator part of the GAN and improves the code coverage. Ganfuzz is still restricted by RNN and unique data structure.

MoWF [24] is based on white-box fuzzing, which uses the file format input model to check the data integrity constraints. Godefroid et al. [25] describes syntax features by using context-independent syntax. Bastani et al. [26] merges non-terminal symbol of context-free grammar by constructing regular expressions to generate testcases. Yang et al. [27], Holler et al. [28], and Veggalam et al.

[29] share the same idea that leveraging context-free grammar. Skyfire [14] solves the problem of highly-structured input with modeling the existing samples by implementing Abstract Syntax Trees (ASTs), then learning a Probability Context Sensitive Grammar (PCSG) that contains grammar features and semantic rules. Learn&Fuzz [5] trains character-level recurrent neural network (CHAR-RNN) which learns the grammar, with PDFs as datasets, to generate highly-structured files. Learn&Fuzz is a grammar-based method which leverages deep learning technique. Learn&Fuzz utilizes RNN to focus on the generation of the grammar itself. However, learning a specific well-formed structure conflicts with the exploring nature of fuzzing [5], which constrains the performance of Learn&Fuzz. Smartseed implements Wassertein-GAN(WGAN) model but does not elaborate on the feasibility of the more sophisticated GAN model [30]. Moreover, they leave the problem of processing highly-structured data to future discussion.

Comparing RapidFuzz with current state-of-art methods as Table 1 shows, AFL [15], AFLFast [19], AFLGo [20] and VUzzer [21] are all mutation-based methods which can not process highly-structured data well. The MoWF [24], GoWF [25], and SkyFire [14] are grammar-based generation fuzzing. They can process part of highly-structured data, but due to the limitation of the grammar-based model, they do not have enough augmentability to be implemented in various types of formats. Learn&Fuzz, FasterFuzzing [6], Smartfuzz [31], and Smartseed [30] are machine-learning based generation methods. The last three do not concentrate on speed and efficiency on highly-structured data. Learn&Fuzz finds a conflict between the learning system and the nature of fuzzing [5]. Hence, a heuristic idea based on the above questions is associating inputs generation with inputs mutation by leveraging a sewing algorithm. The learning-based system that serves as a mutation acceleration part attaches to the main fuzzer. RapidFuzz increases the mutation speed for deeper information hidden in the program. As [32] concludes from an information-theoretic perspective, the information is critical to fuzzing efficiency.

3. RapidFuzz

This paper proposes RapidFuzz based on the insight that GAN generation can be adopted to accelerate the mutation speed, which allows our method to skip part of meaningless mutation. Fuzzing for highly-structured data always consumes excessive computational resources. This section details the workflow of our method, which is organized as the Fig. 1 shows:

3.1. Data pre-processing

Data processing is vital to machine-learning based approach. In our paper, it can be divided into 2 phases: data collection and data conversion. These two phases figure out how to obtain sufficient training data to stabilize the learning model. Data collection solves the problem of data sources for the training set. Data transformation solves the problem of how structured data processed by GAN. Furthermore, data conversion considers the challenge of augmentability, which enables RapidFuzz to be expanded to multiple programs. These two phases are introduced as follows.

As Fig. 1 illustrates, in order to obtain the high-quality merged testcases, the training set needs to utilize a set of seed files evolving correct structure information. From the observation, the quality of the dataset directly influences the performance. We input the initial seed file into the AFL for five hours to generate thousands of mutated seed files. All these seed files not only remain the original structure but also include the mutation information. Generated testcases contain similar distribution to the training dataset, which can be extracted to assist the AFL process.

For different types of highly-structured data, it is necessary to unify the data processing [30]. A uniform method keeps all 6 types of input generation models in the same metric. In comparison, the GAN has specific requirements on the input data format. Thus, we need to unify the training set into the same format that is processed by GAN. Furthermore, ensuring output can be restored to its original file format. In our approach, we read the input as binary flow at first, then convert the binary flow into a numeric matrix via Base64 transcoding. Due to the various file format sizes, it is necessary to select an encoding method to represent all six file formats.

The whole data preprocessing procedure consists of the following steps as Fig. 2 shows: 1) Reading input with the binary flow. 2) Encoding the binary flow with Base64. 3) Converting the encoded codes to Base64 digits. 4) Transforming the digital sequence to the matrix of 108*108.

First, we collect thousands of seed files from AFL, which allows us to obtain a reasonable amount of training data that enough to specify the input feature. These input features refer to the probability distribution characteristics in the matrix. As a result, the generator does not create identical testcases. There are still subtle variations.

Second, we utilize the Base64 as our encoding method. The Base64 has a total of 65 codes, including 52 letters, 10 digits, '+', '/' and '='. The Base64 reads a 3-byte binary sequence each time and converts it into four 6-bit corresponding sequences. According to the various sizes of the experimental files, we have chosen the matrix of 108*108 to ensure the capability. The redundant space is filled with the '=' mark in each corresponding type for the convenience of decoding. This conversion method does ensure the compatibility of RapidFuzz.

3.2. Wassertein Generative Adversarial Networks with Gradient Penalty

One of the most critical insights in this paper is the implementation of GAN as part of our generation model. Though GAN shows

Table 1
The state-of-art fuzzing methods.

Methods	Grammar	Highly-structured	Type	Experiment
AFL	No	No	mutation	–
AFLFast	No	No	mutation	objdump, etc
AFLGo	No	No	mutation	Binutils Diffutils
VUzzer	No	No	mutation	LAVA
FairFuzz	No	No	mutation	readelf, etc
MoWF	Yes	Yes	generation	PNG
GoWF	Yes	Yes	generation	JavaScript
SkyFire	Yes	Yes	generation	XML
Learn&Fuzz	No	Yes	generation	Pdf
FasterFuzzing	No	No	generation	Ethkey
Smartfuzz	Yes	No	generation	XML
Smartseed	No	No	generation	mp3, etc
RapidFuzz	No	Yes	generation	Multiple

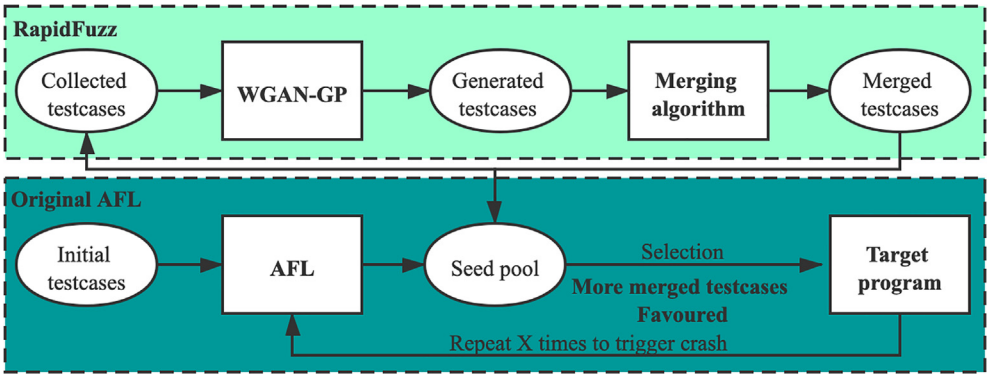


Fig. 1. Approach overview of RapidFuzz.

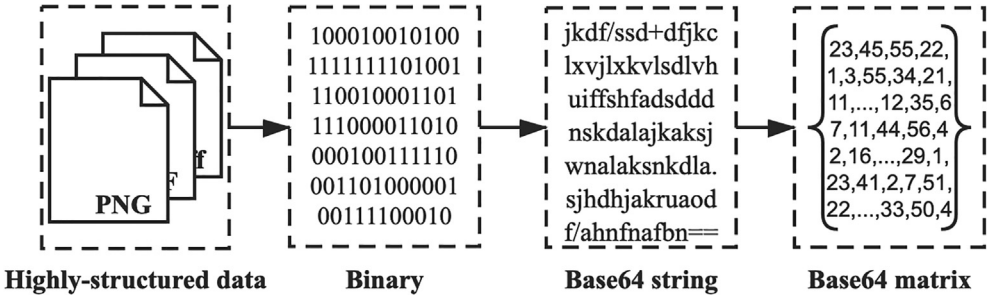


Fig. 2. Workflow of conversion.

its remarkable effectiveness in the computer vision domain, it still has the potential to be dug in other areas. GAN has a fame for generating images based on the observation that pixels change smoothly. A probability density matrix seems reasonable to describe the status of the image. As a matter of fact, similar to the image, RapidFuzz does not requires complete specification in points correspondingly. Because RapidFuzz considers this variation as part of the mutation. Therefore, RapidFuzz utilizes a 108*108 matrix to conduct data preprocess, which makes GAN possible for various data formats. This method is detailed in Section 3.1.

GAN consists of two separated networks, a generator and a discriminator, which represents a zero-sum game [33]. The objective function of original GAN, as shown in Eq. (1), represents the probability that the discriminator considers it to be a real sample or a generated sample. In the training process of GAN, the discriminator hopes to maximize the maximization of the objective function. This architecture does works on generating similar probability distribution matrix, which using loss function like KL divergence to evaluate the effectiveness.

$$\min_G \max_D V(D,G) = E_{x\sim p(x)}[log(D(x))] + E_{z\sim p_z(z)}[log(1-D(G(z)))] \quad (1)$$

where G and D represents discriminator and generator in GAN respectively. The $x \sim p(x)$ denotes the distribution of real data that G wants to learn from. The $z \sim p_z(z)$ is the prior input noise which interrupt learning of G. The equation demonstrates a dynamic balance between G and D. The purpose of the generator is to obtain generated data P_g that has the smallest distance with real data distribution P_{data}. Before WGAN-GP proposed, Wasserstein GAN (W-GAN) improves the GAN model from the perspective of Wasserstein distance. It ameliorates the original GAN from training instability problem. The Wasserstein distance is shown in Eq. (2). The Wassertein distance describes a minimum cost from a distribution to another. Different from the unsatisfied performance of JS divergence and KL divergence, the Wassertein distance can stabilize the training process with relatively smoother measurement [34].

$$W(P_r,P_g) = \sup_{\|f\|_L\le 1} E_{x\sim P_r}[f(x)] - E_{x\sim P_g}[f(x)] \quad (2)$$

where $W(P_r,P_g)$ represents the Wasserstein distance between the distribution of real data P_r and generated data P_g. The occurrence of WGAN dramatically alleviates the problem of unstable training. The weight clipping method introduced in WGAN still leads to unexpected behavior. In order to deal with the optimization difficulty, WGAN-GP proposes a further advanced method to obey the 1-Lipschitz continuity, which can be seen as gradient penalty in loss function. Consequently, WGAN-GP can learn the probability distribution on the original dataset more stably and efficiently than other GAN models.

3.3. Mutation gene detection algorithm

where to mutate?

The mutation gene detection algorithm serves as the sewing algorithm to eject mutation to the original seed pool. In Smartseed, they implement the WGAN as the generation model without modification on the generated testcases. RapidFuzz ranks the frequency of sensitive mutation points(in another word, hot points) in the training set samples obtained from AFL. Leveraging a semi-random approach to ensure that high-frequency mutation points appear in GAN-generated samples will be combined with original AFL-generated samples. The low-frequency mutation point is determined by a random function simultaneously. This process allows the low-frequency mutation points to appear in the total inherited mutation points with a lower probability but still exists, which indicates the genetic inheritance can be obtained from parent generation.

In this algorithm, sensitive mutation points tend to detect those bytes that exist in testcases that are tended to be mutated by AFL. For example, AFL provides information about id_000015,

src_000000, op_flip1, pos_7552 and +cov which indicate the file has a 1-bit flip operation on the 7552nd byte. Therefore, the 7552nd byte serves as a mutation point of the 000000 file. For all of the seed files generated by AFL, if this mutation point shows its high frequency to appear, we mark it as a sensitive mutation point. Different from unstructured data, highly-structured data are more challenging to mutate. By adopting this mutation gene detection algorithm, testcases can absorb those more valuable points with higher probability.

Moreover, AFL always finds arduous to mutate the testcase which can pass the parser for highly-structured data. Through the statistical analysis of the testcases generated in the fuzzing test, we extract those mutation points with high-probability to guide the merging between the two sides of testcases. The pseudo code is shown as below:

Algorithm 1. Mutation Gene Detection Algorithm

```
procedure MUTATIONSELECTION λ, n
    k ~ U(0, 1)                          ▷ Initialize the storing space
    list ← null
    while number of list < n do
        if order(x_i) < n/2 then
            list ← list + x_i v 7:        else
            k ← random(0, 1)
        end if
        if k > λ
            list ← list + x_i
        else
            return
        end if
    end while
end procedure
```

We collect the mutation dataset given by AFL. Then, the frequency of the mutation points is counted, where the high-frequency bytes represent that the byte is not susceptible to the format resolver and has a higher probability of mutating. First, we arrange all the mutation points in descending order. For the lesser half of the mutation points, we generate a random number k obeys $U(0,1)$. If the random number is greater than a given threshold, then the byte is marked as a sensitive mutation point. This method defines the high-frequency mutation point as the sensitive mutation point, and the low-frequency mutation point still has a certain probability of becoming a sensitive mutation point. Therefore, for each sample generated by GAN, a certain proportion of sensitive mutation points are randomly selected from the mutation point dataset, and the value of source file is sewed to the generated sample, thereby increasing the mutation probability of the generated sample.

4. Evaluation

In this section, we conduct the experiment on 9 different open-source programs with 6 types of highly-structured data, including png, tiff, xml, elf, pdf, and jpeg. All six formats are highly-structured formats. For each format, there are average 2–3 programs to prove the effectiveness of our approach. These formats are popular in our daily life, and their potential bugs deeply threaten software security. These 6 types of format conclude various functions. The jsonlint, pngfix, and xmllint are respectively from libjson [35], libpng [36], and libxml2 [37]. The function of tiff2pdf is to convert the tiff file into a pdf file. The tiff2pdf contains a parser for the tiff file. The function of tiffdump is to extract file directory information. Readelf, strip, and objdump are selected from GNU.

For instance, Tiff2pdf is a program that converts tiff files into pdf files. This program contains a parser for tiff files that makes it difficult to fuzz. Tiffdump is used to extract the directory information within the tiff file. As shown in Table 2, the selected target programs include multiple formats and various functions, including format conversion, file reading and etc. This selection strategy ensures that we can verify our method in a comprehensive aspect.

We provide two experiment scenarios to discuss the effectiveness of the method as below:

Scenario 1 First, we run AFL for 5 h to collect the training set, then inject the generated seed into AFL to perform for continuous 24 h as the comparing group. Second, input the generated seed by AFL into RapidFuzz for further mutation to obtain new 300 seeds simultaneously. Third, inject the 300 seeds into AFL for 24 h again as the experimental group.

Scenario 2 Fuzzing process and training process are conducted simultaneously. We obtain the training set for GAN after 6 h run of AFL and FairFuzz. After the training process finished, we inject the merged testcases to AFL (or other fuzzer) in parallel. The whole time limit is set as 24 h.

We believe that the GAN model is recyclable to use. The time consumption of training is negligible with a long period of testing process. In addition, different with conventional fuzzing methods, the training process of deep learning-based systems mostly rely on GPU computational resources, which allows a chance for enhancing fuzzer performance without interrupting its CPU based operations. We limit the number of initial seeds to one for scenario 2, since the testcases generated by Rapidfuzz are added in the middle of fuzzing. A single initial seed with relatively lower initial coverage can demonstrate the effect of the algorithm better.

Our software experiment configuration is set as follows: python 3.6.2, Anaconda3, Keras 2.1.6, tensorflow-gpu 1.8.0, numpy 1.14.0. The hardware configuration includes the Intel i7-8700 K CPU with 32G memory, and the graphics card is TITAN X with 12G memory.

4.1. Evaluation metrics

4.1.1. Mapsize

Mapsize is the coverage calculation method in AFL [15]. AFL allocates an array sharemem[] with 65 KB storing space. Each array element contains one tuple to store the bitmap of the execution path. For the execution path A→B→C→D→E, it is saved in the array as [AB], [BC], [CD], [DE]. When AFL obtains a new testcase, AFL checks its trace of testcase and compares the testcase for their existing paths. For instance, there are two trace paths A→B→C→D→E and A→B→C→A→E. The tuples gain new paths which are [CA] and [CE]. If there is a new path like A→B→C→E, it will not be considered because no new tuple appears. This structure was designed in AFL to avoid the path explosion problem and provide an overall method to evaluate the fuzzing quality in programs.

In our paper, the Fig. 3 and the Table 3 show the performance of RapidFuzz. All the experiments are conducted for 24 h, and RapidFuzz behaves distinguished from AFL in both coverage and speed for the majority of target programs. RapidFuzz achieves higher mapsize which means RapidFuzz discovered more paths than AFL except pngtest. Thus, RapidFuzz discovers more status which hides the potential bugs and crashes. As a matter of fact, RapidFuzz detects 10, 3, and 3 crashes in cjpeg, tiff2pdf, and Strip, while AFL finds 3, 0, and 2.

4.1.2. Code coverage

Code coverage usually refers to those program codes that are triggered by fuzzing inputs [38]. The method provides coverage

Table 2
Experiment programs.

Lib	Name	Version	Format	Parser
Libpng	Pngfix	1.6.36	png	yes
Libtiff	Tiff2pdf	3.6.1	tiff	yes
Libtiff	Tiffdump	3.6.1	tiff	no
Libxml2	Xmllint	2.4.16	xml	yes
Libjson	jsonlint	0.8	json	yes
GNU	Readelf	2.6.1	elf	yes
GNU	Strip	2.6.1	elf	yes
GNU	objdump	2.6.1	elf	yes
libjpeg	cjpeg	6b	jpeg	yes

Table 3
Mapsize result.

Programs	RapidFuzz+AFL	AFL	Enhance
Pngfix	5.61%	4.80%	***16.8%***
Tiff2pdf	2.99%	2.46%	***21.5%***
Tiffdump	0.79%	0.65%	***21.5%***
Xmllint	7.62%	7.61%	0.1%
jsonlint	0.43%	0.39%	***10.2%***
Readelf	10.63%	9.96%	***6.7%***
Strip	0.90%	0.86%	***4.6%***
objdump	0.98%	0.96%	2.0%
cjpeg	2.29%	2.23%	2.7%

metrics from three different aspects, which include function coverage, basic block coverage, and branch coverage. see Fig. 4.

A basic block is a set of instructions that all instructions are executed sequentially and only once. The basic block has an entry that jumps to other basic blocks after all the instructions in the basic block are executed. A function is a block of code in the program with a full capability to complete a task. Each function is designed with different purposes. Function coverage is one of the basic coverage. The branch is two basic blocks that are continuously executed. It can be regarded as a structure composed of two basic blocks and their connecting edges. Branch coverage represents the execution situation between basic blocks in the program.

The code coverage collected in this paper is obtained by angr and qemu, which angr is an advanced symbolic execution framework. Symbolic execution is a kind of method that transforms a program into a control flow graph to find the vulnerabilities. Angr implements multiple symbolic executions and static analysis methods to achieve better performance. Qemu is a mode provided in the AFL to simulate program execution.

RapidFuzz calls the angr interface to obtain the control flow graph and branch information about the tested program. And then calls qemu to obtain the trace information. The branch coverage is calculated from the branch information of trace and the total branch information provided by angr. The basic block coverage is calculated by the number of basic blocks of each testcase provided by qemu and the total basic blocks in the branch provided by angr. Furthermore, the function coverage is calculated from the ratio of function entry provided by qemu and the function entry in the control flow graph provided in angr. The equations are shown as follows:

$$Basic\_block\_coverage = \frac{Basic\_block\_tested}{Total\_basic\_block} \tag{3}$$

$$Function\_coverage = \frac{Functions\_tested}{Total\_functions} \tag{4}$$

$$Branch\_coverage = \frac{Branches\_tested}{Total\_branches} \tag{5}$$

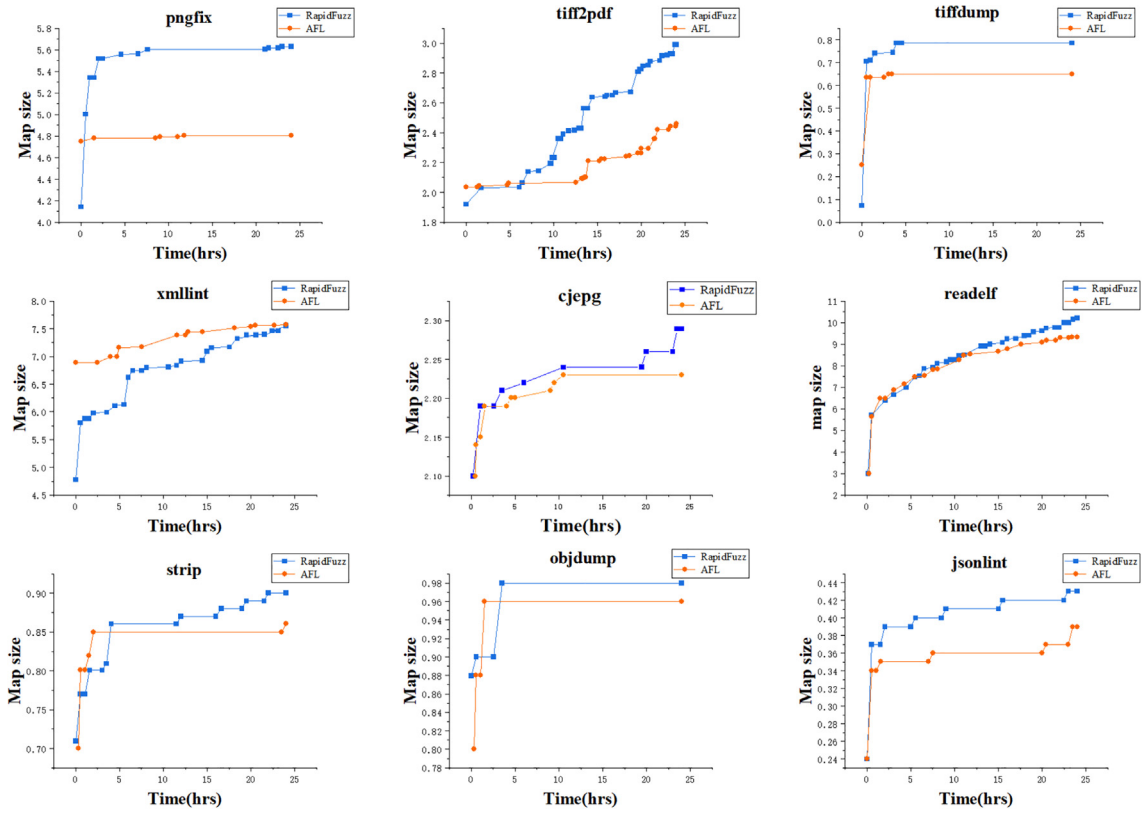


Fig. 3. The performance of RapidFuzz and AFL on 9 real-world programs.

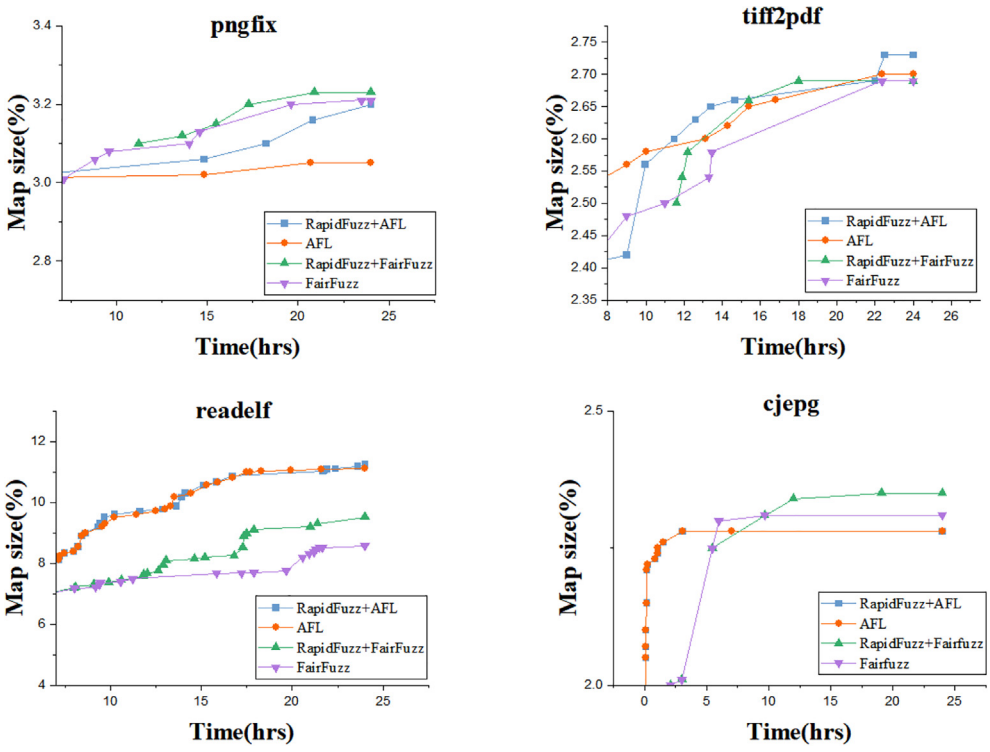


Fig. 4. The comparison experiments with implementation on the top of AFL and FairFuzz.

Comparing with AFL, a reasonable observation is that RapidFuzz enhances the code coverage performance in most of the tested programs. For tiff2pdf and tiffdump, the enhancing rate even exceeds more than 20% which shows in Table 4. On one hand, considering the enhancing percentage, RapidFuzz successfully refined the process of AFL. On the other hand, RapidFuzz does accelerate the entire fuzzing process, which can be observed from the graph 3. In the majority of the tested programs, RapidFuzz has the advantage in all branch coverage, basic block coverage, and function coverage. The Table 4 also confirms our insight that the more radical and extensive mutation brought by GAN improves the quality of the testcases. Further increases the code coverage of the fuzzing test.

For scenario 2, we compare our method with Fairfuzz in two aspects: 1, We compare the version of RapidFuzz based on AFL with FairFuzz; 2, We implement RapidFuzz on the top of the FairFuzz to combine the advantages of two methods to demonstrate the transportability of RapidFuzz.

We find that RapidFuzz works better than Fairfuzz for most of cases. We believe this advantage grows as the testing time increases. In addition, we find that the combination of RapidFuzz +AFL does not achieve better performance in cjpeg. We believe the reason is that the quality of deep-learning models is highly dependent on the training set. The program collects samples after AFL runs for 6 h. We observe that the map size no longer changes after AFL is run for 3 h until the end of the test. The boosting is challenging to achieve any progress in this case.

4.1.3. Favored rate

AFL has its own process to absorb the meaningful testcases. Basically, as much as AFL absorbs, the fuzzing process pushes forward. Finding the testcase that can cover the most branches, and further select to get the final result according to its size and execution time. The Eq. (6) explains the entire process. From the Table 5, for different tested programs, almost 300 testcases generated by RapidFuzz can partially be absorbed by AFL into the favored test-

Table 4
Performance on all tested programs.

Program	Basic block coverage			Function coverage			Branch coverage		
Name	RF+AFL	AFL	ER	RF+AFL	AFL	ER	RF+AFL	AFL	ER
tiff2pdf	1478/2972	1194/2972	24%	91/116	83/116	10%	1986/4996	1564/4996	27%
tiffdump	354/487	296/487	20%	33/40	28/40	18%	505/741	410/741	23%
Pngfix	6527/39924	6460/39924	3%	191/388	191/388	1%	10407/86509	10017/86509	5%
Xmllint	198/4570	198/4570	0%	18/207	18/207	0%	132/9084	132/9084	0%
Readelf	4961/18301	4770/18301	4%	110/224	108/224	2%	6900/32909	6504/32909	6%
Objdump	537/18088	535/18088	0%	60/388	60/388	**0%**	1598/33090	596/33090	0%
Strip	519/11532	515/11532	0%	79/349	78/349	1%	526/21096	522/21096	1%
Jsonlint	337/1601	319/1601	6%	16/48	15/48	7%	475/2993	449/2993	6%
cjpeg	2923/15045	2842/15045	3%	97/136	96/136	1%	4263/30236	4148/30236	3%

R.F. is the abbreviation for RapidFuzz.
E.R. is the abbreviation for Enhancing Rate.

Table 5
Favored Rate.

Programs	Pngfix	Tiff2pdf	Tiffdump	Xmllint	jsonlint	Readelf	Strip	objdump	cjpeg
F.R.	56/300	63/300	41/300	33/300	47/300	42/300	25/300	37/300	43/300

F.R. stands for Favored Rate.

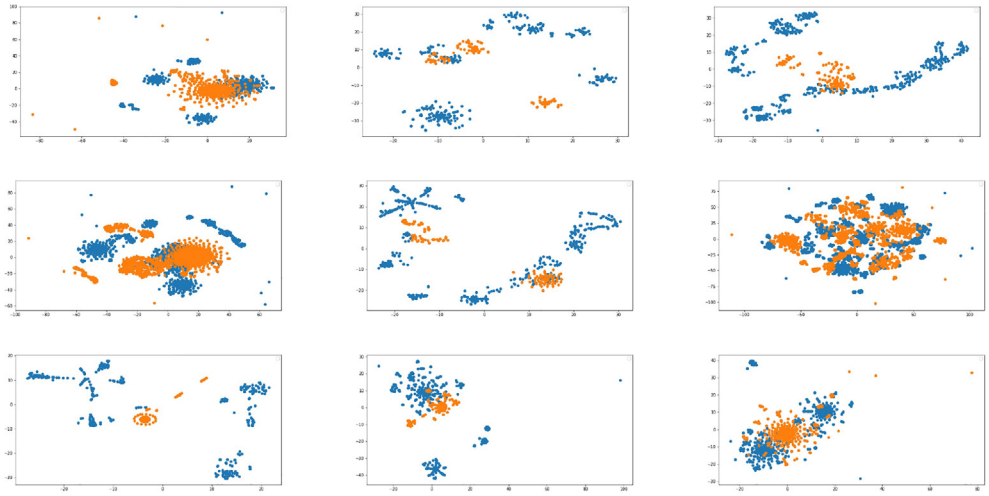


Fig. 5. The two-dimensional distribution of testcases based on t-SNE.

case set. The highest proportion of tiff2pdf reaches 21%, which also echoes the outstanding performance of tiff2pdf in the code coverage experiment. These selected testcases are samples that trigger non-repeating paths. This indicates the degree which testcases are accepted by the fuzzing tool. According to the Table 5, the testcases generated by GAN show the ability to improve the efficiency of AFL after recombination of the mutation gene detection algorithm.

$$favored\_rate = \frac{favored\_testcase}{all\_testcase} \tag{6}$$

4.1.4. t-SNE

T-distributed Stochastic Neighbor Embedding(t-SNE) is a nonlinear dimensionality reduction machine-learning based method. The core idea of t-SNE is to discover a mapping from high-dimensional space to low-dimensional space, which makes it possible to visualize the high-dimensional data. Visualized data can reflect the difference between data in an understandable way. In RapidFuzz, the implementation of t-SNE illustrated the distribution of our merged testcases directly. The distribution of testcases is respectively visualized by t-SNE, as shown in Fig. 5. It is not arduous to figure out the distribution of AFL testcase shows much more concentration than RapidFuzz testcases. This represents in high-dimensional space, the testcases generated by AFL mutates slower than RapidFuzz does. The samples generated by RapidFuzz are separately centered with AFL testcases but still show much-expanded vast distribution than AFL. This indicates that RapidFuzz can produce more meaningful testcase than AFL does, which discover the in-depth status of the tested programs.

5. Conclusion

In this paper, we propose an innovative method named RapidFuzz. The critical insight in this paper is leveraging deep learning techniques and sewing algorithms to accelerate the mutation speed in the original state-of-art fuzzing tool. RapidFuzz can significantly raise the speed to achieve the same coverage in AFL, which also means it can obtain higher coverage simultaneously.

The experiments show, for most of the selected programs, mapsize is improved about 20%, which means RapidFuzz can greatly enhance fuzzer in coverage. Moreover, the implementation of t-SNE illustrates that the mutation rate is impressively raised but still obeys the distribution for most cases. This paper proposes a testcase generation method based on the deep-learning generation model. The quality of testcase generation is highly dependent on the program under test and efficient obtaining of the training set. For future work, we will continue to work on how to obtain the high-quality testcases for the training set automatically. In conclusion, RapidFuzz conducts a reasonable method to combine generation-based fuzzing and mutation-based fuzzing and accelerates fuzzing exceedingly.

CRediT authorship contribution statement

Aoshuang Ye: Writing - original draft. **Lina Wang:** Conceptualization, Supervision, Funding acquisition. **Lei Zhao:** Methodology, Conceptualization, Supervision. **Jianpeng Ke:** Investigation. **Wenqi Wang:** Investigation. **Qinliang Liu:** Software.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

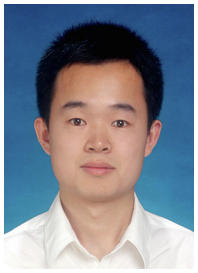
This work was supported by the Project of National Natural Science Foundation of China (61876134), and the Key Project of National Natural Science Foundation of China (U1836112) and (U1536204).

References

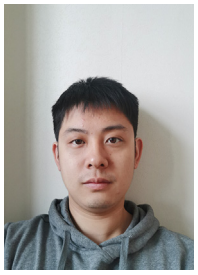
[1] B. Feng, A. Mera, L. Lu, P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling, in: 29th {*USENIX*} Security Symposium {*USENIX*} Security 20), pp. 1237–1254..
[2] M. Böhme, B. Falk, Fuzzing: On the exponential cost of vulnerability discovery, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 713–724..
[3] J. Wang, B. Chen, L. Wei, Y. Liu, Superion: Grammar-aware greybox fuzzing, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp. 724–735..
[4] A. Fioraldi, D.C. D'Elia, E. Coppa, Weizz: Automatic grey-box fuzzing for structured binary formats, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1–13.
[5] P. Godefroid, H. Peleg, R. Singh, Learn&fuzz: Machine learning for input fuzzing, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, pp. 50–59.
[6] N. Nichols, M. Raugas, R. Jasper, N. Hilliard, Faster fuzzing: Reinitialization with deep neural models, CoRR abs/1711.02807 (2017).
[7] Z. Hu, J. Shi, Y. Huang, J. Xiong, X. Bu, Ganfuzz: a gan-based industrial network protocol fuzzing framework, in: Proceedings of the 15th ACM International Conference on Computing Frontiers, pp. 138–145..
[8] E.J. Schwartz, T. Avgerinos, D. Brumley, All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask), in: 2010 IEEE Symposium on Security and privacy, IEEE, pp. 317–331..
[9] C. Cadar, D. Dunbar, D. Engler, Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, 2008, pp. 209–224.
[10] J. Clause, W. Li, A. Orso, Dytan: a generic dynamic taint analysis framework, in: Proceedings of the 2007 International Symposium on Software Testing and Analysis, pp. 196–206..
[11] Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011, The Internet Society, 2011..
[12] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, Z. Chen, Collafl: Path sensitive fuzzing, in: 2018 IEEE Symposium on Security and Privacy (SP), IEEE Computer Society, Los Alamitos, CA, USA, 2018.
[13] B.P. Miller, L. Fredriksen, B. So, An empirical study of the reliability of unix utilities, Communications of the ACM 33 (1990) 32–44.
[14] J. Wang, B. Chen, L. Wei, Y. Liu, Skyfire: Data-driven seed generation for fuzzing, in: 2017 IEEE Symposium on Security and Privacy (SP), 2017, pp. 579–594.
[15] M. Zalewski, American fuzzy lop, URL:http://lcamtuf.coredump.cx/afl/, 2017..
[16] V. Ganesh, T. Leek, M. Rinard, Taint-based directed whitebox fuzzing, in: 2009 IEEE 31st International Conference on Software Engineering, pp. 474–484..
[17] T. Wang, T. Wei, G. Gu, W. Zou, Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection, in: 2010 IEEE Symposium on Security and Privacy, IEEE, pp. 497–512..
[18] F. Jiang, C. Zhang, S. Cheng, Fffuzzer: Filter your fuzz to get accuracy, efficiency and schedulability, in: Australasian Conference on Information Security and Privacy, Springer, pp. 61–79..
[19] M. Böhme, V.-T. Pham, A. Roychoudhury, Coverage-based greybox fuzzing as markov chain, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ACM, New York, NY, USA, 2016, pp. 1032–1043.
[20] M. Böhme, V. Pham, M. Nguyen, A. Roychoudhury, Directed greybox fuzzing, in: CCS 2017 - Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, volume Part F131467, Association for Computing Machinery (ACM), United States, 2017, pp. 2329–2344..
[21] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, H. Bos, Vuzzer: Application-aware evolutionary fuzzing., in: NDSS, vol. 17, pp. 1–14..
[22] C. Lemieux, K. Sen, Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 475–485..
[23] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, X. Zhou, Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit, in: 29th {*USENIX*} Security Symposium {*USENIX*} Security 20), pp. 2307–2324..
[24] V.-T. Pham, M. Böhme, A. Roychoudhury, Model-based whitebox fuzzing for program binaries, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 543–553..
[25] P. Godefroid, A. Kiezun, M.Y. Levin, Grammar-based whitebox fuzzing, in: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 206–215..
[26] O. Bastani, R. Sharma, A. Aiken, P. Liang, Synthesizing program input grammars, ACM SIGPLAN Notices 52 (2017) 95–110.
[27] X. Yang, Y. Chen, E. Eide, J. Regehr, Finding and understanding bugs in c compilers, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 283–294..
[28] C. Holler, A. Zeller, K. Herzig, Fuzzing with code fragments, Proc Usenix Security (2012) 445–458.
[29] S. Veggalam, S. Rawat, I. Haller, H. Bos, Ifuzzer: An evolutionary interpreter fuzzer using genetic programming., in: European Symposium on Research in Computer Security..
[30] C. Lv, S. Ji, Y. Li, J. Zhou, J. Chen, P. Zhou, J. Chen, Smartseed: Smart seed generation for efficient fuzzing, CoRR abs/1807.02606 (2018).
[31] D. Molnar, X.C. Li, D.A. Wagner, Dynamic test generation to find integer bugs in x86 binary linux programs, in: Proceedings of the 18th Conference on USENIX Security Symposium, USENIX Association, Berkeley, CA, USA, 2009, pp. 67–82.
[32] M. Böhme, V.J. Manès, S.K. 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, pp. 678–689..
[33] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, Generative adversarial nets, in: Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, K.Q. Weinberger (Eds.), Advances in Neural Information Processing Systems 27, Curran Associates Inc., 2014, pp. 2672–2680.
[34] M. Arjovsky, S. Chintala, L. Bottou, Wasserstein generative adversarial networks, in: D. Precup, Y.W. Teh (Eds.), Proceedings of the 34th International Conference on Machine Learning, volume 70 of Proceedings of Machine Learning Research, International Convention Centre, Sydney, Australia, 2017, pp. 214–223.
[35] Libjson, URL: https://sourceforge.net/projects/libjson/, 2020..
[36] Libpng, URL: http://www.libpng.org, 2020..
[37] Libxml2, URL: http://www.xmlsoft.org/, 2020..
[38] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, et al., Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers, in: 30th USENIX Security Symposium (USENIX Security 21). USENIX Association..

Aoshuang Ye received the B.E. degree from Jianghan University, China and M.S. degree from Japan Advanced Institute of Science and Technology, Japan in 2015 and 2018, respectively. He is currently pursuing his Ph.D. degree in Information Security with the School of Cyber Science and Engineering, Wuhan University. His research interests focus on AI stesting and verification.

Lina Wang received the B.S. degree from Hefei University of Technology, Hefei, China, in 1986, and the M.S. and Ph.D. degrees from Northeastern University, Shenyang, China, in 1989 and 2001, respectively. She is a professor in School of Cyber Science and Engineering, Wuhan University, China, and the member of China Computer Federation. Her main research interests include multimedia security, cloud computing security, and network security.



Lei Zhao received the BSc and PhD degree in computer science and engineering both from Wuhan University, China, in 2007 and 2012, respectively. He is an associate professor at the School of Cyber Science and Engineering, Wuhan University, China. His research interests include software testing and debugging, program analysis, and software security.



Jianpeng Ke received the B.E. degree from Xidian University and M.S. from Wuhan University, China in 2016 and 2018, respectively. He is currently pursuing his Ph.D degree in Cyberspace Security with the school of Cyber Science and Engineering, Wuhan University. His research interests focus on AI security.

Wenqi Wang received the B.E. and M.S. degree from NanKai University, China, in 2015 and 2017, respectively. He is currently pursuing his Ph.D. degree in Information Security with the School of Cyber Science and Engineering, Wuhan University. His research interests focus on AI security.

Qinliang Liu received the B.E. degree from Hubei University and M.S. from Wuhan University. His current research interest is about big data analysis.