SmartSeed: Smart Seed Generation for Efficient Fuzzing

Chenyang Lv, Shouling Ji, Yuwei Li, Junfeng Zhou, Jianhai Chen, Pan Zhou, Jing Chen

Abstract—Fuzzing is an automated application vulnerability detection method. For genetic algorithm-based fuzzing, it can mutate the seed files provided by users to obtain a number of inputs, which are then used to test the objective application in order to trigger potential crashes. As shown in existing literature, the seed file selection is crucial for the efficiency of fuzzing. However, current seed selection strategies do not seem to be better than randomly picking seed files. Therefore, in this paper, we propose a novel and generic system, named *SmartSeed*, to generate seed files towards efficient fuzzing. Specifically, `SmartSeed` is designed based on a machine learning model to learn and generate high-value binary seeds. We evaluate `SmartSeed` along with American Fuzzy Lop (AFL) on 12 open-source applications with the input formats of mp3, bmp or flv. We also combine `SmartSeed` with different fuzzing tools to examine its compatibility. From extensive experiments, we find that `SmartSeed` has the following advantages: First, it only requires tens of seconds to generate sufficient high-value seeds. Second, it can generate seeds with multiple kinds of input formats and significantly improves the fuzzing performance for most applications with the same input format. Third, `SmartSeed` is compatible to different fuzzing tools. In total, our system discovers more than twice unique crashes and 5,040 extra unique paths than the existing best seed selection strategy for the evaluated 12 applications. From the crashes found by `SmartSeed`, we discover 16 new vulnerabilities and have received their CVE IDs.

Index Terms—Fuzzing, vulnerability detection, seed generation.

I. INTRODUCTION

An application bug refers to an error, failure or fault in a computer system's application (such as an operating system or an image browser) that causes it to behave wrong. There are several common application bug types such as buffer overflow, integer overflow, use-after-free and so on.

Application bugs can cause enormous harms. An application bug of Therac-25 resulted in the deaths of several patients [12]. On June 4, 1996, an Ariane 5 rocket exploded just forty seconds after its lift-off. It was caused by an application error and lost the cargo which is worth $500 million [16]. In 2002, a study by the National Institute of Standards and Technology (NIST) estimated that application bugs costed U.S. $59.5 billion loss a year [21]. Thus, application bugs can cause both security problems and financial loss. It is important to

C. Lv, S. Ji, Y. Li, J. Zhou and J. Chen are with the College of Computer Science and Technology, Zhejiang University, China. E-mail: {puppet, sji, liyuwei, zhoujf620, chenjh919}@zju.edu.cn.

P. Zhou is with the School of Electronic Information and Communications, Huazhong University of Science and Technology, Wuhan, Hubei, 430074, China. E-mail: panzhou@hust.edu.cn.

Jing Chen is with the College of Computer Science and Technology, Wuhan University, China. E-mail: chenjing@whu.edu.cn.

find and fix application bugs before they cause accidents or are exploited by attackers.

Over the decades, extensive research has been spent for application bug detection. There are several existing staple techniques in the academia and industry community, such as dynamic taint analysis, symbolic execution and fuzzing.

Dynamic taint analysis is one of the most popular methods to detect application bugs [13], [14]. It marks the data from untrusted inputs as tainted. Then, the analysis tool traces the tracks of all the tainted data when the application is running. If the tainted data is used in some dangerous memory, the analysis considers that a bug is detected. Dynamic taint analysis has a low false positive rate by discovering bugs in this way. However, since the analysis needs to add necessary checks and traces the track of data when the application is running, dynamic taint analysis has low execution efficiency.

Symbolic execution is a method of detecting bugs by considering the whole process as solving constraints [22], [23]. It regards the input of an objective application as variable X and tries to figure out the relationship between X and the execution paths of the application. If a symbolic execution tool runs into a branch statement, it will record two constraints, one for each branch. Thus, when symbolic execution is finished, users have a bunch of constraints, each of which corresponds to an execution path of the objective application. If users want to analyze one of the paths, they can solve the corresponding constraint to get the corresponding X. In other words, users obtain the corresponding input to test the target execution path. Symbolic execution can accurately locate the bugs in an application. However, it has a poor scalability. When the objective application is large, the final equation becomes too complicated to be solved for.

Fuzzing is one of the most effective methods to detect crashes in applications. Different from other kinds of methods, fuzzing dose not analyze which data or which code results in the crashes. A fuzzing tool uses a simple yet efficient way to detect crashes. It generates a great number of input files. Then, the tool tests the objective application with these inputs and detects whether the application behaves abnormally. If the objective application happens to have a crash, the fuzzing tool will store the input file that triggers the crash. In this way, the fuzzing tool can detect the crashes and obtain the input files which trigger crashes. Then, users can study these input files to figure out whether the objective application has bugs. However, since fuzzing does not consider how to trigger crashes in an objective application, the whole process of detecting crashes is blind. Fortunately, with the help of current powerful computing capability, a fuzzing tool can use a large

number of input files to test an objective application so many times within a short time. Therefore, fuzzing can potentially find more crashes. In this paper, we focus on providing a better seed set for fuzzing in order to discover more crashes.

Nowadays, modern application is much larger and more complex, which makes it harder to adapt the fuzzing tools as needed. To improve fuzzing efficiency, researchers typically develop strategies for improvement following two directions: design better fuzzing tools and use a better fuzzing seed set.

Following the first direction, there are a number of works focusing on designing better tools or improving current fuzzing tools.

The fuzzing tools based on generation rules can learn the input format of the objective application [6], [15], [19], [20], [26]. Then, they can generate highly-structured input files based on the input format. While other fuzzing tools spend most time passing the format check, generation-based fuzzing tools use the generated highly-structured files to test the execution of applications. Thus, the generation-based fuzzing tools are better at detecting the crashes of applications which check the syntax features and the semantic rules of the input files.

The fuzzing tools based on genetic algorithms do not consider the input format of the objective application. According to genetic algorithms, the tools mutate the initial input seed set by byte flipping, cross over and so on. Then, in order to discover crashes, the tools take the initial input seed set and the mutated input files as the inputs of the applications. Since they require little prior knowledge of the objective application, the mutation-based fuzzing tools work pretty efficiently. However, sometimes they get stuck because of the simple crash detection strategy. To improve the efficiency of mutation-based fuzzing tools, a number of researches combine the fuzzing tools with other vulnerability detection technologies such as static analysis, taint analysis and symbolic execution [4], [5], [7], [17], [18]. There are several other researches indicating that stimulating fuzzing tools to improve the coverage and test low-frequency paths can improve the efficiency of fuzzing [2], [3], [39], [40], where new fuzzing tools based on their viewpoints are presented respectively.

The second direction of improving the efficiency of fuzzing tools is to use a better seed set. Allen and Foote presented an algorithm to consider the parameter selection and automated selection of seed files [25]. The algorithm was implemented in an open-source Basic Fuzzing Framework (BFF). Woo et al. developed an analytic framework to evaluate 26 randomized online scheduling algorithms that schedule a better seed to fuzz a program [24]. In 2014, Rebert et al. evaluated six seed selection strategies of fuzzing and showed how to select the best selection strategy [1], etc.

However, the current seed selection strategies have many deficiencies. For example, some strategies require a lot of time to obtain the seed set [1]. What's more, proved by our experiments and the abovementioned work [1], the current seed selection strategies perform unstable in many application scenarios. Further, they do not have evident advantages than random seed selection in many cases.

Therefore, to solve the problem of how to obtain a better seed set for the applications without highly-structured input format, we come up with the following heuristic questions:

Q1: Can we generate valuable seeds in a fast and effective manner? As we discussed before, many existing seed selection strategies are slow, which is improper for fuzzing. More importantly, existing solutions cannot yield effective seeds in many scenarios. Therefore, to address these limitations, instead of studying how to select seeds, our primary goal is to study how to automatically generate effective seeds in a fast manner leveraging state-of-the-art machine learning techniques.

Q2: Can we generate valuable seeds in a robust manner? Based on the assumption that we have already figured out a fast and effective seed generation strategy, it is still inefficient if `SmartSeed` can only generate valuable seeds for some specific input format or we have to retrain the model everytime we want to fuzz a new application with the same input format. Thus, our second goal is to design a robust seed generation system. It should be able to generate valuable seeds for multiple input formats. Moreover, we only need to train the model once for any kind of input format. Then, the files generated by this model can improve the fuzzing performance for other applications with this input format.

Q3: Can we generate valuable seeds in a compatible manner? It is unexpected for most fuzzing cases if our system can only generate valuable seeds for specific fuzzing tools. Therefore, to improve the compatibility, we aim to design a seed generation strategy that can combine with different fuzzing tools and improve their performance.

Following the above heuristic questions, in this paper, we present a novel seed generation system named *SmartSeed* to provide fuzzing tools with a better seed set as shown in Fig. 1.
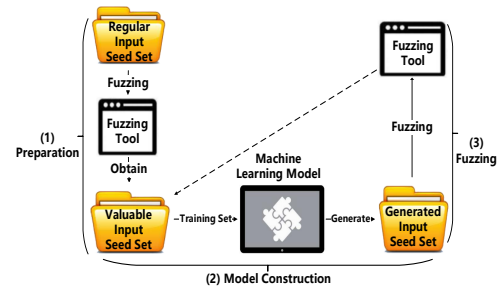

Fig. 1. Workflow of `SmartSeed`.

Basically, the workflow of `SmartSeed` consists of three stages.

(1) **Preparation.** `SmartSeed` is a machine learning-based system. To bootstrap `SmartSeed`, we need to prepare necessary training data. Specifically, we collect some regular files and employ them to fuzz some applications by commonly used fuzzing tools like American Fuzzy Lop (AFL) [11]. Then, we collect the input files that trigger unique crashes or new paths as the training data. Note that, this step is only for collecting necessary training data to bootstrap `SmartSeed` (which can then be used to generate seeds for fuzzing many applications), and can be easily implemented in practice. We show the details in Section II-B.

(2) **Model Construction.** To make `SmartSeed` easily extendable in practice, we propose a transformation mechanism to encode the raw training data into generic matrices, which are then being employed to construct a generative model for seed generation. Leveraging the generative model, we generate effective files as seeds.

(3) **Fuzzing.** Leveraging the seeds generated from the constructed generative model, we use fuzzing tools (e.g., AFL) to discover crashes of objective applications.

Note that the whole process can form a closed-loop. The machine learning model can generate effective seed files to help fuzzing tools discover new crashes and paths of objective applications. Then, the training set of the machine learning model in `SmartSeed` can be refined and enhanced with complementary input files that trigger new crashes or paths.

With the help of `SmartSeed`, users can efficiently generate a valuable input seed set. Our evaluation on 12 open source applications demonstrates that `SmartSeed` significantly improves the performance of fuzzing compared to state-of-the-art seed selection strategies. The main contributions in this paper can be summarized as follows:

• We present a machine learning-based system named `SmartSeed` to generate valuable binary seed files for fuzzing applications without of requiring highly-structured input format.

• Combining with AFL, we evaluate the seed files generated by `SmartSeed` on 12 open source applications with the input formats such as mp3, bmp or flv. Compared with state-of-the-art seed selection strategies, `SmartSeed` finds 608 extra unique crashes and 5,040 extra new paths than the existing best strategy in total.

• We further combine `SmartSeed` with other popular fuzzing tools to examine its compatibility: (1) combining with AFLFast [2], `SmartSeed` is still the best seed strategy for each application. (2) `SmartSeed` + honggfuzz [29] finds the most crashes for five of the six objective applications. (3) Among the evaluated seed selection/generation strategies, only `SmartSeed` discovers crashes on `ps2ts` and `mp42aac` when using VUzzer [3] as the fuzzing tool.

• We further analyze the seed sets generated by `SmartSeed` and other state-of-the-art seed selection strategies, and present several interesting findings to enlighten the research of fuzzing. Visualized by t-SNE [30], the seed files generated by `SmartSeed` are closer to the most valuable files that trigger crashes or paths. Meanwhile, the files of `SmartSeed` that trigger unique crashes cover the largest area, which implies that the generated files are easier to be mutated into more discrete valuable files. What's more, we realize that the execution speed is an improper indicator for discovering crashes. However, the larger generation of seeds helps fuzzing tools discover more unique paths. In total, `SmartSeed` finds 23 unique vulnerabilities on 9 applications, including 16 undiscovered ones. In the end, we open source the code of `SmartSeed`, which is expected to facilitate future fuzzing research [1].

---

[1]We also make our system publicly available to facilitate the research in this area, which is available at: https://github.com/puppet-meteor/SmartSeed.

The remainder of this paper is organized as follows. In Section II, we describe the detail of `SmartSeed`. We evaluate `SmartSeed` and existing state-of-the-art seed selection strategies on 12 applications, combine `SmartSeed` with different fuzzing tools and analyze the results of vulnerabilities discovered by different seed selection strategies in Section III. In Section VIII, we further analyze the performance of different seed selection strategies. In Section V, we make some discussions and remark the limitation of our work. We conclude the paper in Section VI. We provide more evaluation results and summarize the related work with remarks in the *Supplementary File*.

II. SmartSeed

A. System Architecture

The core idea of `SmartSeed` is to construct a generative model. Then, we use this model to fast generate valuable files as the input seed set of fuzzing tools.
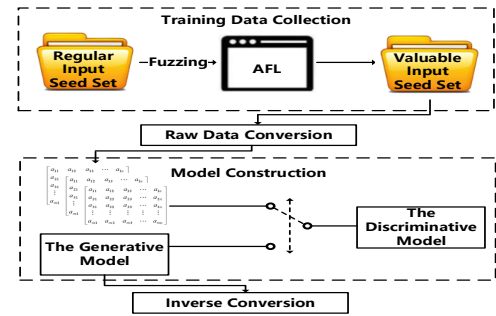

Fig. 2. Architecture of `SmartSeed`.

As shown in Fig. 2, the whole architecture of `SmartSeed` can be divided into 4 procedures:

(1) **Training data collection:** We introduce a criterion to measure the value of input files and present a method to obtain a training set for `SmartSeed` (Section II-B).

(2) **Raw data conversion:** To deal with files with unfixed formats or unfixed file sizes, we convert the binary files of raw training data to a uniform type of matrices (Section II-C).

(3) **Model construction:** Taking the matrices as training data, we construct a seed generative model based on Wasserstein Generative Adversarial Networks (Section II-D).

(4) **Inverse conversion:** Based on the generative model, we generate new matrices and convert them into proper input files, which is the reverse process of Procedure (2) (Section II-E).

In our system, the employed fuzzing tool can be flexible, i.e., `SmartSeed` can be combined with most existing mutation-based fuzzing tools. Since AFL is one of the most efficient existing fuzzing tools [11], by default, we select AFL to be the fuzzing tool in our implementation.

B. Training Data Collection

To construct a machine learning model for generating valuable seed files, we need to obtain an initial training set first.

Certainly, we are expected to ensure that an input file in the training set is really valuable. Otherwise, `SmartSeed` may

not be able to learn useful features of "valuable input files" and further generate such kind of files. Therefore, we first clarify valuable input files. Specifically, in our implementation, we define valuable files as the input files that trigger unique crashes or unique paths of applications. The reasons are as follows: (1) since the ultimate goal of fuzzing is to detect more crashes, the input files are considered as valuable if they can trigger unique crashes of objective applications; (2) according to the existing research [2], [3], [39], [40], increasing the coverage and the depth of fuzzing paths are more likely to increase the number of explored crashes. Hence, the files triggering new paths are also valuable from this perspective.

Intuitively, we may employ existing seed selection strategies to select a few valuable input files as the training set. However, according to the existing research [1] and the results of our experiments, current seed selection strategies seem to be unreliable. Then, we realize that fuzzing tools such as AFL will store the input files that trigger unique crashes or paths, which faultlessly coincides with our needs. Thus, we have the following training data collection strategy: we can first use regular input files collected from the Internet to fuzz the applications with the same input format. Then, we gather the valuable input files, which trigger unique paths or unique crashes of those applications, as the training set of `SmartSeed`.

Note that, the above training set construction is not a limitation of `SmartSeed` in practice. First, the criterion for determining a valuable input file is that it can trigger a crash or a new path, which can be easily implemented by AFL in practice. Second, for the applications with the same input formats, such as the applications that deal with mp3, we can fuzz multiple applications simultaneously to facilitate the training set construction process, and meanwhile increase the multiformity of the input files in the training set. Third, since we can parallelly run many fuzzing procedures, we can further accelerate the training set collection process, e.g., we can collect more than 20,000 valuable files for one kind of input format within a week. Therefore, we can construct the training set for `SmartSeed` easily and fast in practice.

By collecting a training set in this way, we further have the following advantages:

• We can accurately evaluate the value of the input files. The input files in the training set can certainly detect unique paths or trigger unique crashes of some applications. Thus, they carry useful features for learning.

• During the fuzzing process, we realize that the formats of many files that trigger crashes or new paths are corrupted. We analyze the reason is that files are randomly mutated according to the employed genetic algorithm, and it seems that the corrupted files are more likely to trigger unique crashes and paths. However, it is difficult to gather a number of corrupted files from the Internet, while `SmartSeed` can be trained to generate many corrupted files as expected.

C. Raw Data Conversion

To construct a generic seed generation model, we propose a mechanism to convert the raw input files in the training set to a uniform type of matrices. The reasons for conducting such conversion are as follows.

First, one of our objectives is to make `SmartSeed` deal with multiple input formats and unfixed file sizes. However, it is inconvenient to adjust the data read mode for different kinds of and different sizes of files. Thus, we should figure out a uniform method to read data from the training set. Second, the formats of many files in the training set are corrupted. A normal reading manner, such as reading a bmp picture file as a three-dimensional matrix, may not work in many application scenarios. Third, based on the knowledge that machine learning algorithms are better to work on quantitative values of matrices rather than some random value types, we are expected to find a way that can convert multiple types of files to a uniform type of matrices. Finally, we expect to give expression to the magic bytes in the binary form of training files. Because in this way a machine learning model would be easier to learn the features of magic bytes that control the code execution paths. Thus, the files should be read in binary form and are expected to be converted to uniform matrices.

Below, we introduce the main procedures of raw training data conversion, which are shown in Fig. 3.

(1) Since all the files can be read in binary form, we can read the binary form of any type of file and get a binary string.

(2) To handle the problem of how to recognize the end of the binary string, we encode the string with Base64. Thus, we have the string formed by 64 kinds of characters and "=", e.g., the *character string* shown in Fig. 3.

(3) Now, since a character string may have 65 different characters, for convenience, we convert the characters of Base64 and "=" to the numbers (from 0 to 64). Thus, we obtain a string of numbers as shown in Fig. 3. The encoding mechanism for such conversion is shown in Fig. 4.

(4) To economize the number of elements in a matrix, we convert every six numbers of the number string to a large number of the decimal system. Then, we normalize the numbers of the decimal system to [0, 0.75418890624] (since $65^6 - 1 = 75,418,890,624$) for the accuracy and efficiency of the model training. Finally, add 0 at the end of the matrix if there is an empty element, as shown in Fig. 3.

Since the larger numbers will be stored in the scientific notation form and lose some veracity, one element of the matrix can only store one number in the decimal system that is converted from six numbers (from 0 to 64) at most. However, users can choose the number less than six if the file size is small in practical application scenarios. Note that the fewer numbers are converted into one large number, the easier a machine learning model learns the features of input files. We convert 6 numbers in our experiments in order to test the most difficult situation for the machine learning model.

Leveraging the above method, we can convert raw files with corrupted formats or unfixed file sizes into a uniform type of matrices. This can significantly improve the extendability and compatibility of `SmartSeed`.

D. Model Construction

One of the best existing generative models is the Generative Adversarial Networks (GAN) model, which has been

Fig. 3. Training Data Conversion.

Fig. 4. Convert Base64 Character to the Number from 0 to 64.

widely used for unsupervised learning since 2014 [8]. GAN is a new framework consisting of a generative model and a discriminative model. The generative model tries to generate fake data that are similar to the real data in the training set, while the discriminative model tries to distinguish the fake data from the real data. Two models alternatively work together to train each other and further to improve each other. As a result, the generative model will generate data that are too real to be distinguished by the discriminative model. Usually, the generative model provided by GAN can generate more realistic data than other algorithms. However, it is unstable to train a GAN model. GAN also has many problems such as model collapse.

Radford et al. tried multiple combinations of machine learning schemes to construct better generative and discriminative models for GAN. They presented the Deep Convolutional GAN (DCGAN) model [9]. DCGAN can generate more realistic data than standard GAN and is much easier for training. However, it still has many problems including model collapse.

In 2017, Arjovsky et al. presented the Wasserstein GAN (WGAN) model [10]. Unlike other GAN models, WGAN improves the stability of learning a lot. It is also much easier to train a WGAN model. In addition, WGAN can solve the problems of GAN like mode collapse in most application scenarios.

Hence, for our purpose, we employ WGAN to learn the characteristics of valuable files and then generate valuable seed files. Our selection further has the following benefits. First, WGAN can learn the features of the training set by itself. Thus, we do not need to pay attention to the feature selection, which is very time-efficient. Another advantage is that we can freely choose an appropriate machine learning model as the generative model and the discriminative model of WGAN. We analyze that Multi-Layer Perceptron (MLP) focuses more on every quantitative value in a matrix, while Convolutional Neural Network (CNN) pays more attention to

the global features of a matrix. Then, in order to construct a better model, we test the performance of both neural network models. For our application, MLP does work better than CNN as the generative and discriminative models of WGAN. It also requires less training time. Therefore, we choose MLP as the model in `SmartSeed`. Now, based on the collected training data, we can train a generative model for valuable seed generation.

Note that, a detailed description of WGAN can be found in [10]. Since our focus in this paper is to construct a GAN-based efficient fuzzing framework and further demonstrates its effectiveness, we leave the research of developing an improved GAN model as the future work. Furthermore, users of `SmartSeed` can select an alternative machine learning model as the generative model in terms of the application scenarios.

E. Inverse Conversion

In this subsection, we introduce how to employ the generative model of `SmartSeed` to obtain an effective input seed set. Since the training set of `SmartSeed` is a number of matrices, the generative model is trained to generate similar matrices. To obtain binary input files for fuzzing, we have to convert the generated matrices to binary files. Thus, we do the reverse of the abovementioned procedures in Section II-C.

To be specific, the fist step is to restore the [0, 0.75418890624] elements of the matrix to large numbers of the decimal system. Second, convert each number of the decimal system to six numbers (from 0 to 64), i.e., the *number string* as shown in Fig. 3. Then, convert the numbers (from 0 to 64) to the characters of Base64 and "=", i.e., the *character string* as shown in Fig. 3. Finally, decode the character string of Base64 into a binary file and store it locally. Thereout, we obtain the input files for the following fuzzing.

III. IMPLEMENTATION & EVALUATION

A. Datasets

To evaluate the performance of `SmartSeed`, we employ 12 open-source linux applications with input formats of mp3, bmp or flv as shown in Table I.

We choose these applications mainly for three reasons.

First, they can be fuzzed with the files without high-structured formats, which are the focus of genetic algorithm-based fuzzing as well as `SmartSeed`.

Second, these applications are popular and important open-source programs. For instance, `mp3gain` is a popular tool to analyze and adjust the volume of mp3 files. From *SourceForge*

[32], `mp3gain` for linux are downloaded 639 times on average per week and `mp3gain` for Windows are downloaded 10,877 times on average per week. `magick` is the main program of `ImageMagick`, whose mirror has been collected 1.7k times on *github*. `bmp2tiff` is the conversion tool of `libtiff`, which is employed in the experiments of many other researches as a fuzzing dataset [31]. As for flv, `avconv` and `ps2ts` are the popular media tools provided by `libav` and `tstools`, respectively.

Third, these applications have different code logic and functionalities, and thus are sufficiently representative. For example, `magick` is used to browse bmp, `sam2p` is used to convert bmp into eps, `bmp2tiff` is used to convert bmp into tiff, and `exiv2` is a cross-platform C++ library and a command line utility to manage image metadata. All the four applications are provided by different groups. Although `mp42aac` is a tool in `Bento4` that deals with mp4 files, we also use `SmartSeed` to generate flv files as the initial seed set to fuzz it. This is mainly for evaluating the robustness of `SmartSeed`.

B. SmartSeed Implementation

To implement `SmartSeed`, the first step is to collect the training set. As we mentioned before, we fuzz some applications and collect the valuable input files.

In consideration of the training efficiency, if the size of training files is too large, we have to use a big matrix to store the file, which leads to longer training time. On the contrary, it is hard to collect small files for meaningful multimedia data with formats such as mp3 and flv. What's more, small files may not carry enough features for machine learning. Therefore, we need to determine a proper size for the employed training files.

Considering both the training efficiency and the training effectiveness, we prefer to employ files less than 17KB to construct the generative model of `SmartSeed`. To accommodate files with size of 17KB or less, 64×64 matrices are sufficient (whose maximum storage is 18KB). On the other hand, it would be hard for `SmartSeed` to learn meaningful features from too sparse matrices, e.g., a matrix with more than 35% of its elements are null, since they carry less valuable information. Therefore, we finally decide to collect valuable input files with size between 12KB and 17KB as the training

TABLE I
OBJECTIVE APPLICATIONS.

	Target	Version	Release Time	Heat
mp3	mp3gain	1.5.2.r2	2010-08	639 downloads/week
	ffmpeg	3.4	2018-02	10.6k collections
	mpg123	1.25.6	2017-08	1,720 downloads/week
	mpg321	0.3.2	2012-03	78 downloads/week
bmp	magick	7.0.7	2017-10	1.7k collections
	bmp2tiff	3.8.2	2007-03	52 collections
	exiv2	0.26	2017-05	67 collections
	sam2p	0.49.4	2017-12	10 collections
flv	avconv	12.2	2017-09	576 collections
	flvmeta	1.2.1	2016-08	67 collections
	ps2ts	1.13	2015-11	70 collections
	mp42aac	1.5.1.0	2017-09	443 collections

data in our implementation. This setting can ensure that the training rate is fast, make it easy to collect valuable files and meanwhile ensure sufficient information is carried, i.e., seek a balance between efficiency and utility.

Note that easy extendability is one of `SmartSeed`'s advantages. The size of the input files in the training set is adjustable in practice. If users want to use smaller files as the training set, they may convert fewer characters of Base64 to the number of the decimal system or use smaller matrices. On the other hand, if users want to use files with bigger size to train `SmartSeed`, they may employ a bigger matrix to store the string of Base64.

Now, we are ready to collect the training data. The collection processes are conducted on 16 same virtual machines with an Intel i7 CPU, 4.5GB memory and a Ubuntu 16.04 LTS system and last for a week for each input format. For applications with the mp3 format, we employ AFL to fuzz `mp3gain` and `ffmpeg`, and collect 20,646 valuable input files with size 12KB to 17KB as the training set. For bmp, we use AFL to fuzz `magick` and `bmp2tiff`. Then, we collect 24,101 input files as the training set. For applications with the flv format, we employ AFL to fuzz `avconv` and `flvmeta`. Then, we collect 21,688 input files between 12KB and 17KB that discover new paths or unique crashes.

Based on the collected training data, we construct the prototype of `SmartSeed`. The core function of `SmartSeed` is implemented in Python 2.7. The code of the WGAN model is implemented in Pytorch 0.3.0. We use Adam as the optimization algorithm of the WGAN model. Then, we test several times to decide the hyper-parameters, such as how many times to train the discriminative model after training the generative model. During the training process, we decrease the learning rate of the optimization algorithm from 0.5×10^{-3} to 0.5×10^{-12} inch by inch. It takes us around 38 hours to train the generative model of `SmartSeed` for 100,000 times on a server with 2 Intel Xeon E5-2640 v4 CPUs running at 2.40GHz, 64 GB memory, 4TB HDD and a GeForce GTX 1080TI GPU card. Note that others may not need to train the models for 100,000 times.

C. Effectiveness

1) Comparison Strategies: To compare the performance of `SmartSeed` with state-of-the-art seed selection strategies, we implement the following methods.

`random`. Under this scheme, we randomly select files from the regular input files that are usually downloaded from the Internet. Because this is the most common seed selection strategy in practice, we take `random` as the baseline seed strategy in our experiments.

`AFL-result`. Under this scheme, we randomly select seeds from the saved input files of AFL, which are also the files used for training `SmartSeed`. Since these files are certainly to either trigger unique crashes or new paths during the fuzzing process, they may yield an outstanding performance as a seed set for fuzzing tools. As our system is learnt from these files, `AFL-result` can serve as the baseline of `SmartSeed`.

`peachset`. The fuzzing framework Peach provides a seed selection tool named `MinSet` [28], whose workflow is as

follows: (1) `MinSet` inspects the coverage of each file in the full set which might be collected from the Internet; (2) it sorts the files by their coverage in the descending order; (3) `MinSet` initializes the seed set as an empty set, which is denoted by `peachset`; (4) `MinSet` checks the coverage of files in order. If a file improves the coverage of `peachset`, the file will be added to `peachset`.

`hotset`. `hotset` contains the files that discover the most unique crashes and paths within t time [1]. To construct `hotset`, first, use each file, that may be collected from the Internet, as the initial seed set to fuzz an application for t seconds. Second, sort the files in the descending order by the number of discovered crashes and paths. Third, select the top-k files to constitute `hotset`. In our experiments, we fuzz each file for 240 seconds.

`AFL-cmin`. In order to select a better seed set, AFL provides a tool named `AFL-cmin`. The core idea of `AFL-cmin` is to filter out the redundant files that inspect already discovered paths. `AFL-cmin` tries to find the smallest subset of the full set, which still has the same coverage as the full set.

2) Results: Now, we evaluate the effectiveness of different seed generation/selection strategies. For the schemes that need an initial dataset to bootstrap the seed selection process, we collect a dataset consisting of 4,600 input files for each input format from the Internet. For `SmartSeed`, we generate seeds based on its generative model. All the following experiments are conducted on same machines with an Intel i7 CPU, 4.5GB memory and a Ubuntu 16.04 LTS system.

Seed Generation Speed. First, we evaluate the seed generation speed of different strategies. The results are shown in Table II, from which we have the following observations.

(1) As we discussed before, `random` simply picks a seed set at random from the initial dataset. `AFL-result` follows a similar strategy except for randomly picking a seed set from the saved input files of AFL. Therefore, they are the fastest ones for selecting seeds and are scalable.

(2) `SmartSeed` employs a generative model to generate seeds which is also very fast and scalable. It only takes 12 seconds to generate 100 seed files and 240 seconds to generate 2,000 seed files, exhibiting a linear increasing correlationship between the time consumption and the number of generated seeds.

(3) For the coverage-based seed selection strategies `peachset` and `AFL-cmin`, they are relatively slow. For instance, to generate 500 seed files, it takes `peachset` and `AFL-cmin` 2,500 seconds and 874 seconds, respectively. This is mainly because they have to check the coverage of each file. Furthermore, since coverage-based strategies are hard to increase the number of the selected files by increasing the size of the initial file set, we have to use the screening tools to filtrate multiple initial file sets containing different files to obtain enough selected files.

(4) For `hotset`, it is extremely slow. For instance, `hotset` requires >2,000 min to select seeds from 500 files in our experiments. This is mainly because `hotset` has to use each single file as the seed set to fuzz an application for sufficient time, which is very time consuming.

In practical fuzzing applications, O(100) number of seeds are usually sufficient to bootstrap fuzzing tools. For instance, 3 seed files are sufficient for bootstrapping VUzzer by default [3]. Therefore, based on the results in Table II, `SmartSeed` is very efficient in generating seeds.

Fuzzing Effectiveness. Now, we evaluate the effectiveness of the seeds obtained by different strategies. In our experiments, we employ each seed generation/selection strategy to obtain a seed set consisting of 100 files and then feed the seed set to AFL. Note that it is sufficient and effective to use 100 files as the seed set to guide fuzzing tools, which is analyzed in detail in the *Supplementary File* due to the space limitations. To control irrelevant variables, we operate all the fuzzing experiments on the same virtual machines for 72 hours with an Intel i7 CPU, 4.5G memory and a Ubuntu 16.04 LTS system.

The primary goal of fuzzing is to discover crashes. Thus, we use the number of unique crashes found by each seed strategy as the main evaluation criterion. Furthermore, as shown in many existing researches [2], [3], [39], [40], a higher coverage can improve the fuzzing performance. Therefore, our second evaluation criterion is the number of discovered unique paths during the same time. The results are shown in Table III. We can deduce the following conclusions from Table III.

(1) For discovering unique crashes, `SmartSeed` is very effective and performs the best in almost all the evaluation scenarios. When fuzzing mp3 applications, `SmartSeed` discovers 24 more crashes than the existing best solution for `mp3gain`, and discovers more than twice unique crashes than the existing best solution for `mpg123` and `mpg321`. When fuzzing bmp applications, `SmartSeed` also yields the best performance. Again, when fuzzing flv applications, `SmartSeed` discovers the most crashes in total among the evaluated solutions. An exception is `flvmeta`, on which `AFL-cmin` discovers the most crashes. After viewing the the saved valuable files of `flvmeta`, we conjecture the reason is that normal flv files are more likely to find crashes of `flvmeta`, while `SmartSeed` tends to generate corrupted files that are more likely to find paths of `flvmeta`, since many of its training data are corrupted.

(2) For triggering unique paths, again, `SmartSeed` is very effective. Among the 12 evaluated applications, `SmartSeed` discovers the most paths on 8 ones, the second-most paths on one and the third-most paths on three ones. Specifically, for mp3, `SmartSeed` discovers nearly twice unique paths than other fuzzing strategies except for `ffmpeg`. It also outperforms other strategies when fuzzing `magick`, `bmp2tiff`

TABLE II
SEED GENERATION SPEED.

seed strategy	100 seeds	500 seeds	2,000 seeds
SmartSeed	12 sec	60 sec	240 sec
random	0.5 sec	0.5 sec	0.5 sec
AFL-result	0.5 sec	0.5 sec	0.5 sec
peachset	500 sec	2,500 sec	10,000 sec
hotset	>400 min	>2,000 min	>8,000 min
AFL-cmin	174.8 sec	874 sec	3,496 sec

TABLE III

UNIQUE CRASHES AND PATHS OF EACH OBJECTIVE APPLICATION DISCOVERED BY DIFFERENT FUZZING STRATEGIES.

	SmartSeed+AFL		random+AFL		AFL-result+AFL		peachset+AFL		hotset+AFL		AFL-cmin+AFL	
	unique crashes	unique paths	unique crashes	unique paths	unique crashes	unique paths	unique crashes	unique paths	unique crashes	unique paths	unique crashes	unique paths
mp3gain	**153**	**1,742**	87	936	128	876	129	841	119	989	95	698
ffmpeg	0	1,592	0	**1,925**	0	1,671	0	1,129	0	1,178	0	1,306
mpg123	**78**	**2,154**	0	1,183	0	1,001	0	1,405	0	1,172	0	1,589
mpg321	**204**	**1,060**	40	766	13	187	37	748	16	128	72	441
magick	**238**	**3,374**	0	697	0	1,149	0	196	0	722	0	265
bmp2tiff	**56**	**714**	21	466	34	684	32	534	21	583	21	498
exiv2	**66**	1,549	20	1,413	38	**2,293**	55	1,096	57	1,593	27	1,202
sam2p	**50**	**1,322**	21	468	36	719	25	520	28	479	12	363
avconv	0	**4,315**	0	1,873	0	4,191	0	1,994	0	1,976	0	2,200
flvmeta	90	1,259	68	886	87	**1,295**	100	1,104	98	1,013	**104**	1,128
ps2ts	**43**	1,692	4	1,381	14	1,740	26	1,472	7	1,419	19	**1,742**
mp42aac	**118**	**658**	80	329	102	585	84	571	53	338	70	453
total	**1,096**	**21,431**	341	12,323	452	16,391	488	11,610	399	11,590	420	11,885
average	**91.3**	**1,785.9**	28.4	1,026.9	37.7	1,365.9	40.7	967.5	33.25	965.8	35	990.4

and `sam2p`. Both `SmartSeed` and `AFL-result` perform well when fuzzing flv applications.

(3) Interestingly, for `mpg123` and `magick`, `SmartSeed + AFL` is the only one that can discover crashes among the evaluated strategies. Specifically, it discovers 78 unique crashes on `mpg123` (mainly bus errors) and 238 crashes on `magick` (mainly segmentation faults). This again suggests that `SmartSeed` is very effective in practice. Moreover, for `ffmpeg` and `avconv`, no strategy finds any crash. We conjecture the reasons are: these two applications are very robust and/or our fuzzing time might be too short to trigger crashes.

(4) For `random`, `AFL-result`, `peachset`, `hotset` and `AFL-cmin`, it is difficult to say which is better. In total, the unique crashes and paths discovered by them are around [340, 490] and [11,000, 16,500], respectively. `peachset` seems to perform better than others in more scenarios. However, even for the naive solution `random`, it outperforms others in many applications. For instance, `random` discovers more crashes than `peachset` and `AFL-result` for `mpg321`, discovers more crashes for `mpg321` and `mp42aac` and the same number of crashes for `bmp2tiff` compared with `hotset`, and discovers more crashes for `sam2p` and `mp42aac` and the same number of crashes for `bmp2tiff` compared with `AFL-cmin`, which is unexpected.

In summary, compared with state-of-the-art seed selection strategies, `SmartSeed` is more stable and yields much better performance. In total, `SmartSeed + AFL` discovers 124.6% more unique crashes (i.e., 608 extra unique crashes) and 30.7% more unique paths (i.e., 5,040 extra unique paths) than the existing best seed strategy.

We also visualize the growth of unique crashes for each strategy. The results are shown in Fig. 5, from which we have the following observations.

(1) `SmartSeed` is very efficient, e.g., when fuzzing `mpg321`, the unique crashes discovered by `SmartSeed` in 10 hours are more than the crashes that discovered by other schemes in 72 hours.

(2) For existing seed selection strategies, they exhibit similar performance during the fuzzing processes and their curves usually mix together. This result is also consistent with the results shown in Table III.

(3) As indicated by the results in Fig. 5, it takes time for each curve to become stable, which implies that sufficient time is necessary to enable these strategies to find crashes. This observation is also meaningful for us to conduct fuzzing evaluation in a proper manner.

In summary, compared with state-of-the-art seed selection strategies, `SmartSeed` performs pretty better than other seed strategies for generating valuable files with multiple input formats. It not only can discover unique crashes faster, but also can discover more.

Number of Seeds vs. Fuzzing Performance. To figure out the relationship between the number of seeds and fuzzing performance, we conduct the comparative experiments in the *Supplementary File* due to the space limitations. The results show there is no distinct connection between the number of seeds and fuzzing performance, while it is OK to use 100 seeds as the initial set to guide fuzzing tools in our experiments.

D. Compatibility

In this subsection, we examine the compatibility and extendibility of `SmartSeed`. Specifically, we combine `SmartSeed` with existing popular fuzzing tools and evaluate its performance.

1) Fuzzing Tools: In this evaluation, in addition to AFL, we consider the following fuzzing tools.

`AFLFast [2]`. AFLFast is a fuzzing tool based on AFL. By using a power schedule to guide the tool towards low-frequency paths, AFLFast can detect much more paths with the same execution counts than AFL.

`honggfuzz [29]`. honggfuzz is an easy-to-use fuzzing tool provided by *Google*. Similar to AFL, honggfuzz modifies the input files from the initial seed set and use them for fuzzing. In addition, honggfuzz provides powerful process state analysis by leveraging *ptrace*.

`VUzzer [3]`. VUzzer is a fuzzing tool that focuses on increasing the coverage. By prioritizing the files mutated from the input files that reach deep paths, VUzzer can explore more
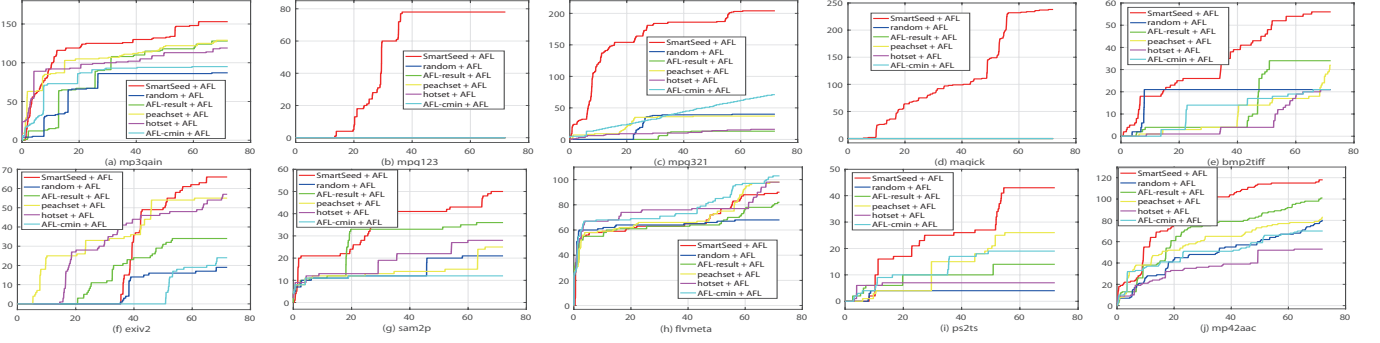
Fig. 5. Number of crashes over 72 hours. X-axis: time (over 72 hours). Y-axis: the number of unique crashes.

and deeper paths. Unlike AFL, AFLFast and honggfuzz who count the edge-coverage, VUzzer uses a dynamic binary instrumentation tool named *PIN* to calculate the block coverage, i.e., VUzzer computes the percent of discovered unique blocks to measure the coverage of an objective application.

2) Results: For comparison with `SmartSeed`, we also consider to combine `random` and `AFL-result` with the considered fuzzing tools. Specifically, we first employ each seed strategy to obtain 100 seed files, and then feed each fuzzing tool with these seeds. For the objective applications, we use `mpg123`, `mpg321`, `magick`, `sam2p`, `ps2ts` and `mp42aac`. All the fuzzing evaluations last for 72 hours and are conducted on same virtual machines with the same settings as in Section III-C.

We show the results in Table IV. Note that, since honggfuzz failed to build `sam2p` and `mp42aac` by its compiler `hfuzz-cc`, we cannot count the discovered unique paths for these two applications when we use honggfuzz to fuzz them. From Table IV, we have the following observations.

(1) With respect to the discovered unique crashes, when combining with AFLFast, `SmartSeed + AFLFast` discovers the most crashes in all the evaluation scenarios. When combining with honggfuzz, `SmartSeed` discovers over twice of crashes on `mpg321` than the other strategies and is the only one that guides honggfuzz to discover crashes on `magick`. While on `mpg123`, `sam2p` and `mp42aac`, all the three seed strategies perform similar. When combining with VUzzer, `SmartSeed` discovers the most crashes on `mpg321`, `sam2p`, `ps2ts` and `mp42aac`. As for `mpg123` and `magick`, no crash is discovered by the evaluated strategies. All the above results demonstrate that `SmartSeed` is compatible with existing popular fuzzing tools, and meanwhile is very effective in fuzzing.

(2) With respect to the discovered unique paths (or coverage when combining with VUzzer), `SmartSeed` also yields the best performance in most of the evaluation scenarios. Specifically, `SmartSeed` discovers the most unique paths in all the cases when combining with AFLFast. As for honggfuzz, except of the error cases (`sam2p` and `mp42aac`), `SmartSeed` discovers more unique paths than others for `mpg123` and `ps2ts`. For `mpg321`, all the strategies discover the same number of paths. When combining with VUzzer, we use the criteria as in [3], i.e., we count the block coverage rate instead

of unique paths. From the results, all the strategies discover similar number of paths on mp3 and bmp applications, while `SmartSeed` yields a much better performance on flv applications.

Overall, `SmartSeed` exhibits a good compatibility when combing with different fuzzing tools. Meanwhile, the seeds generated by `SmartSeed` can significantly improve the performance of existing popular fuzzing tools.

E. Vulnerability Results

To figure out unique vulnerabilities, we recompile the evaluated applications with `AddressSanitizer` [50] and use the discovered files of `SmartSeed`, which triggered crashes, to test the applications. The results are shown in Table V, from which we can learn the following observations.

(1) Although we only run each fuzzing process for 72 hours, from the crashes discovered by `SmartSeed`, we find 23 unique vulnerabilities in total, and 16 of them are previously unreported. We submit them to *CVE* [49] and have acquired the corresponding CVE IDs. This proves that our system is not only efficient but also can guide fuzzing tools to find more undiscovered vulnerabilities.

(2) In total, we discover 9 types of vulnerabilities. This demonstrates that our system does not limit on specific kinds of vulnerabilities, while `SmartSeed` can guide fuzzing tools to discover various vulnerabilities.

In summary, `SmartSeed` is efficient at discovering various types of vulnerabilities in practice. Note that recently, the state-of-the-art fuzzer CollAFL [31] also has fuzzed `exiv2` with the same version and `mpg123` with a close version (ours is 1.25.6, while they used `mpg123` with version 1.25.0) for 200 hours, and discovered 13 and 1 new vulnerabilities, respectively. Nevertheless, we can still discover new and unreported vulnerabilities on these two applications using `SmartSeed` with only 72 hours of fuzzing. This implies that `SmartSeed` is effective to guide fuzzing tools to find undiscovered vulnerabilities.

Then, to evaluate the ability of different fuzzing strategies on discovering unique vulnerabilities, we use input files of all six fuzzing strategies that trigger crashes to test the applications recompiled by `AddressSanitizer` [50]. Since no fuzzing strategy discovers any crash on `ffmpeg` or `avconv` and the assignment team of *CVE* [49] dose not regard the crash

TABLE IV
UNIQUE CRASHES AND PATHS OF EACH OBJECTIVE APPLICATION USING DIFFERENT FUZZING TOOLS.

	mpg123		mpg321		magick		sam2p		ps2ts		mp42aac	
	unique crashes	coverage	unique crashes	coverage	unique crashes	coverage	unique crashes	coverage	unique crashes	coverage	unique crashes	coverage
SmartSeed + AFL	**78**	**2,154**	**204**	**1,060**	**238**	**3,374**	50	**1,322**	43	1,692	**118**	**658**
random + AFL	0	1,183	40	766	0	697	21	468	4	1,381	80	329
AFL-result + AFL	0	1,001	13	187	0	1,149	36	719	14	**1,740**	102	585
SmartSeed + AFLFast	148	**2,526**	161	758	14	2,543	59	**1,611**	179	2,026	86	**592**
random + AFLFast	0	1,333	27	215	1	1,567	27	574	75	1,779	68	569
AFL-result + AFLFast	0	1,241	12	194	0	1,090	53	724	158	1,787	48	360
SmartSeed + honggfuzz	0	**2,776**	22	375	3	5,442	10	error	3	**1,132**	1	error
random + honggfuzz	1	534	9	375	0	4,707	9	error	0	1,069	0	error
AFL-result + honggfuzz	0	534	4	375	0	**5,948**	8	error	3	1,070	1	error
SmartSeed + VUzzer	0	10%	**1,590**	17%	0	39%	16	19%	3	14%	30	**8%**
random + VUzzer	0	**12%**	1,261	**18%**	0	39%	7	**19%**	0	8%	0	1%
AFL-result + VUzzer	0	**12%**	1,483	17%	0	39%	2	18%	0	8%	0	1%

TABLE V
VULNERABILITIES FOUND BY SMARTSEED.

target	type	vulnerability
mp3gain	global-buffer-overflow	CVE-2017-12911; CVE-2017-14410; CVE-2018-10781(+); CVE-2018-10783(+); CVE-2018-10784(+);
mp3gain	segmentation violation	CVE-2017-14406;
mp3gain	stack-buffer-overflow	CVE-2018-10777(+);
mp3gain	memcpy-param-overlap	CVE-2018-10782(+);
mpg123	integer overflow	CVE-2018-10789(+);
mpg321	heap-buffer-overflow	CVE-2018-10786(+);
magick	memory leak	CVE-2017-11754;
bmp2tiff	heap-buffer-overflow	CVE-2018-10779(+);
bmp2tiff	segmentation violation	CVE-2014-9330;
exiv2	heap-buffer-overflow	CVE-2017-17723; CVE-2018-10780(+);
exiv2	stack-overflow	CVE-2017-14861;
sam2p	heap-buffer-overflow	CVE-2018-10792(+); CVE-2018-10793(+);
ps2ts	heap-buffer-overflow	CVE-2018-10787(+); CVE-2018-10788(+);
mp42aac	memory access violation	CVE-2018-10791(+);
mp42aac	buffer overflow	CVE-2018-10790(+); CVE-2018-10785(+);

on `flvmeta` as a vulnerability, we do not show the results of them. The results are shown in Table VI, from which we can learn the following observations.

(1) For the already existed CVEs, all the six fuzzing strategies discover nearly the same number of vulnerabilities on `mp3gain` and `bmp2tiff`, while they do not find any discovered vulnerability on `mpg123`, `mpg321`, `sam2p` and `mp42aac`. Since there is no published CVEs for `ps2ts` or `tstools`, no strategy finds any discovered CVE on `ps2ts`. Only `SmartSeed` finds a discovered vulnerability on `magick`, while others do not find any crash. Although the number of discovered CVEs on `mp3gain` found by `random` and `AFL-result` is one less than the others, they perform pretty well on discovering the found CVEs on `exiv2` and find two more than others. As a conclusion, all the six strategies perform close on discovering the existed CVEs, with `SmartSeed`, `random` and `AFL-result` find the most, while others find one or two fewer.

(2) For undiscovered CVEs, `SmartSeed` finds 16 undiscovered vulnerabilities, which is six more than the second most number discovered by `random` and `AFL-result`. However, the number of undiscovered CVEs found by `peachset`, `hotset` and `AFL-cmin` is 7, 8 and 7, respectively. `SmartSeed` performs pretty well on `mp3gain`, for which it finds three more undiscovered vulnerabilities than others. Only `AFL-result` finds an undiscovered CVE on `bmp2tiff` and another on `ps2ts` that are not discovered by `SmartSeed`.

In summary, `SmartSeed` performs the best on finding both discovered and undiscovered vulnerabilities. In total, we discover 23 CVEs from the 1,096 unique crashes found by `SmartSeed` (including 6 more undiscovered CVEs than others). Although `peachset`, `hotset` and `AFL-cmin` require more time to select seed files and discover more crashes than the baseline `random`, they yield the worst performance on finding unique vulnerabilities, which is unexpected.

IV. FURTHER ANALYSIS

To figure out the reasons why `SmartSeed` performs better than other seed selection strategies, we employ t-SNE [30], which is one of the best dimensionality reduction algorithms that can aggregate similar data together, to visualize the distribution. We analyze the similar distribution of the input seeds generated by different seed selection strategies and the mutated files from these seeds, which is a new angle to understand the fuzzing performance (more details are given in the *Supplementary File*).

A. Execution Count

For most, if not all, of existing popular coverage-based greybox fuzzing tools like AFL, they are designed to prioritize mutating the seed files that are executed fast. Such design is based on the intuition that a fast-executed seed is more likely to be mutated into input files that are also executed fast, and thus, more tests can be conducted on the objective application within a fixed time, followed by more potential crashes might be found. To measure the execution speed of a seed set (or input files on average), we may use the *execution count*, which

TABLE VI
VULNERABILITIES FOUND BY SIX FUZZING STRATEGIES.

	SmartSeed + AFL	random + AFL	AFL-result + AFL	peachset + AFL	hotset + AFL	AFL-cmin + AFL
mp3gain	CVE-2017-12911; CVE-2017-14406; CVE-2017-14410; CVE-2018-10781(+); CVE-2018-10783(+); CVE-2018-10784(+); CVE-2018-10782(+); CVE-2018-10777(+);	CVE-2017-12911; CVE-2017-14410; CVE-2018-10781(+); CVE-2018-10783(+);	CVE-2017-12911; CVE-2017-14410; CVE-2018-10783(+);	CVE-2017-12911; CVE-2017-14407; CVE-2017-14410;	CVE-2017-12911; CVE-2017-14407; CVE-2017-14410; CVE-2018-10777(+);	CVE-2017-12911; CVE-2017-14407; CVE-2017-14410;
mpg123	CVE-2018-10789(+);					
mpg321	CVE-2018-10786(+);	CVE-2018-10786(+);	CVE-2018-10786(+);	CVE-2018-10786(+);	CVE-2018-10786(+);	CVE-2018-10786(+);
magick	CVE-2017-11754;					
bmp2tiff	CVE-2014-9330; CVE-2018-10779(+);	CVE-2014-9330; CVE-2018-10779(+);	CVE-2014-9330; CVE-2018-10779(+); CVE-2018-10801(+);	CVE-2014-9330; CVE-2018-10779(+);	CVE-2014-9330; CVE-2018-10779(+);	CVE-2014-9330; CVE-2018-10779(+);
exiv2	CVE-2017-14861; CVE-2017-17723; CVE-2018-10780(+);	CVE-2017-11339; CVE-2017-14861; CVE-2017-17723; CVE-2018-9145;	CVE-2017-11339; CVE-2017-14861; CVE-2017-14863; CVE-2017-17723;	CVE-2017-14861; CVE-2017-17723;	CVE-2017-14861; CVE-2017-17723;	CVE-2017-14861;
sam2p	CVE-2018-10792(+); CVE-2018-10793(+);	CVE-2018-10792(+);	CVE-2018-10792(+);	CVE-2018-10792(+);	CVE-2018-10792(+);	CVE-2018-10792(+);
ps2ts	CVE-2018-10787(+); CVE-2018-10788(+);	CVE-2018-10787(+); CVE-2018-10788(+);	CVE-2018-10787(+); CVE-2018-10802(+);	CVE-2018-10787(+); CVE-2018-10788(+);	CVE-2018-10787(+); CVE-2018-10788(+);	CVE-2018-10787(+); CVE-2018-10788(+);
mp42aac	CVE-2018-10791(+); CVE-2018-10790(+); CVE-2018-10785(+);	CVE-2018-10791(+); CVE-2018-10790(+); CVE-2018-10785(+);	CVE-2018-10791(+); CVE-2018-10790(+); CVE-2018-10785(+);	CVE-2018-10791(+); CVE-2018-10790(+);	CVE-2018-10791(+); CVE-2018-10790(+);	CVE-2018-10791(+); CVE-2018-10790(+);

is defined as the number of testing/execution times conducted by the fuzzing tool within a time period. Evidently, a larger execution count implies more input files can be mutated from the seeds to test the application and faster execution speed on average for each input file.

Now, under the same settings as the experiments in Table III, we analyze the execution count of the seeds generated by `SmartSeed` and state-of-the-art seed selection strategies. The results are shown in Table VII. Considering the results in Tables III and VII together, we find that: although `SmartSeed + AFL` is the most effective strategy in most of the evaluation scenarios with respect to discovering both crashes and paths, it does not have the largest execution count in most of the cases, i.e., `SmartSeed + AFL` discovers more crashes and paths using fewer input files. For instance, `SmartSeed + AFL` is the only strategy that discovers crashes on `magick` while its execution count is the least one. Therefore, based on the results in Tables III and VII, we conclude that *there is no explicit correlation between the execution count and the number of crashes and paths being discovered.*

The above analysis leads to an interesting insight. *It might be a misunderstanding that seed files with faster execution speed can discover more unique crashes or paths.* Although these seeds can generate more files and test the objective application more times, most of generated files may execute the known paths and thus may discover nothing. Therefore, *valuable seeds are more desired for efficient fuzzing rather than fast-executed ones.*

B. Generation Analysis

For genetic algorithm based fuzzing, we usually use *generation* to measure the mutation/generation-relationship between an input file and the seed files. For instance, the generation of the initial seed files is "1", the files that are mutated/generated from the seed files have a generation of "2", and similarly, the files that are mutated/generated from the files with generation k have a generation of $k+1$.

Then, under the same settings as the experiments in Table III, we show the largest generation number among all the generated files of each seed strategy in Table VII. Considering the results in Tables III and VII together, we find that in general, a large achieved generation implies a better coverage performance. In 11 of the evaluated 12 applications except for `avconv`, the most unique paths are discovered by the fuzzing strategy that has the largest or the second largest generation. For instance, the largest achieved generation of `SmartSeed` is much larger than other seed generation/selection strategies on `mp3gain`, `mpg123`, `magick`, `bmp2tiff`, `sam2p` and `mp42aac`, and meanwhile, `SmartSeed` has a better coverage performance on these applications.

Therefore, based on our results and analysis, we have another interesting insight: it is very likely that *there is a positive correlation between the largest achieved generation and the coverage performance of seed strategies.* This can explain the significant coverage improvement of `SmartSeed`.

Note that many researchers do a lot of work to increase the coverage in order to find more crashes. Now we learn the fact that a better seed set can also improve the coverage.

V. DISCUSSION

In this section, we make some discussions on `SmartSeed` starting from the three proposed heuristic questions in Section I. Then, we remark the limitation and future work of this paper.

Q1: Can we generate valuable seeds in a fast and effective manner? From Table II, the seed generation process of `SmartSeed` is linearly scalable and fast. It only takes

TABLE VII
EXECUTION COUNT AND GENERATION OF EACH OBJECTIVE APPLICATION UNDER DIFFERENT FUZZING STRATEGIES.

execution count /generation	mp3					bmp				flv		
	mp3gain	ffmpeg	mpg123	mpg321	magick	bmp2tiff	exiv2	sam2p	avconv	flvmeta	ps2ts	mp42aac
Smartseed + AFL	**39.9M/15**	12M/2	49.5M/26	50.5M/22	**6.5M/11**	**3.8M/8**	45.9M/14	**4.5M/8**	**20.7M/3**	498.1M/27	340M/23	**26.7M/21**
random + AFL	34.5M/3	**22.9M/3**	56.1M/3	40.1M/26	**24.8M/3**	41.5M/3	57.5M/14	33.9M/3	20.5M/3	338.9M/17	231M/21	5.2M/17
AFL-result + AFL	32M/3	14.7M/**3**	58.5M/3	37.7M/10	15.7M/2	25.8M/4	**73.2M/22**	**74.3M/3**	11.7M/3	454.7M/27	**366M/21**	9.4M/18
peachset + AFL	55.7M/3	8.9M/2	45.2M/3	47.1M/13	21.7M/3	37.5M/4	27.7M/16	44.5M/3	13.3M/3	441.7M/27	228M/22	10.9M/19
hotset + AFL	**59.9M/3**	9.4M/2	**80.2M/3**	42.4M/4	24.1M/4	32.2M/4	34.8M/18	56.3M/3	11.8M/3	421.3M/25	152M/20	1.9M/13
AFL-cmin + AFL	38.4M/3	9.7M/2	66.9M/3	**61.8M/4**	16.8M/3	**48.5M/3**	61.3M/9	41.9M/2	17.8M/3	**664.3M/29**	343M/23	3.1M/13

tens of seconds to generate sufficient seed files for popular fuzzing tools. Furthermore, as we evaluated in Section III, `SmartSeed` significantly outperforms existing state-of-the-art seed selection strategies when fuzzing multiple kinds of practical applications. Therefore, as expected, `SmartSeed` can generate effective seeds in a fast and effective manner.

Q2: Can we generate valuable seeds in a robust manner? As shown in Section II, `SmartSeed` is designed as a generic system to generate valuable seeds for applications without of requiring highly-structured formats. We also implement this system to generate seeds for applications with different formats. As shown in Section III, once the generative model of `SmartSeed` is constructed, it can be employed to fuzz many applications with the same/similar format, and meanwhile, its performance significantly outperforms state-of-the-art seed selection techniques. Therefore, based on our evaluation, `SmartSeed` can generate valuable seeds in a robust manner.

Q3: Can we generate valuable seeds in a compatible manner? As shown in Section III, `SmartSeed` is easily compatible with existing popular fuzzing tools. Furthermore, their fuzzing performance can also be improved in most of the scenarios. Therefore, `SmartSeed` is easily extendable and compatible.

Due to the space limitations, we show the limitation and future work of this paper in the *Supplementary File*.

VI. CONCLUSION

In this paper, we present a novel unsupervised learning system named *SmartSeed* to generate valuable input seed files for fuzzing. Compared with state-of-the-art seed selection strategies, `SmartSeed` discovers 608 extra unique crashes and 5,040 extra unique paths when we use AFL to fuzz 12 open source applications with different input formats. Then, we combine `SmartSeed` with different fuzzing tools. The evaluation results demonstrate that `SmartSeed` is easily compatible and meanwhile very effective. To further understand the performance of `SmartSeed`, we make more analysis on the seeds generated by `SmartSeed` and present several interesting findings to enlighten our knowledge on effective fuzzing. What's more, we find 23 unique vulnerabilities on 9 applications by `SmartSeed` and have obtained CVE IDs for 16 undiscovered ones. Finally, we will open source the `SmartSeed` system, which is expected to facilitate future fuzzing research.

REFERENCES

[1] Rebert A, Sang K C, Grieco G, et al. Optimizing seed selection for fuzzing[C]. *Usenix*, 2014:861-87

[2] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain[C]. *CCS*, 2016, PP(99):1-1.

[3] Rawat S, Jain V, Kumar A, et al. VUzzer: Application-aware Evolutionary Fuzzing[C] *NDSS*. 2017.

[4] Sang K C, Woo M, Brumley D. Program-Adaptive Mutational Fuzzing[C] *S&P*, 2015:725-741.

[5] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution[C] *NDSS*. 2016.

[6] Wang J, Chen B, Wei L, et al. Skyfire: Data-Driven Seed Generation for Fuzzing[C] *S&P*, 2017:579-594.

[7] Haller I, Slowinska A, Neugschwandtner M, et al. Dowsing for overflows: a guided fuzzer to find buffer boundary violations[C]. *Usenix*. 2013:49-64.

[8] Goodfellow I J, Pouget-Abadie J, Mirza M, et al. Generative adversarial nets[C] *NIPS*. 2014:2672-2680.

[9] Radford A, Metz L, Chintala S. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks[J]. *Computer Science*, 2015.

[10] Arjovsky M, Chintala S, Bottou L. Wasserstein GAN. 2017.

[11] American Fuzzy Lop (AFL), *http://lcamtuf.coredump.cx/afl/*.

[12] Therac-25 accidents, *https://en.wikipedia.org/wiki/Therac-25*.

[13] Clause J, Li W, Orso A. Dytan: a generic dynamic taint analysis framework[C].*ISSTA*. 2007:196-206.

[14] Kang M G, Mccamant S, Poosankam P, et al. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation[C]. *NDSS*. 2011.

[15] Godefroid P, Kiezun A, Levin M Y. Grammar-based whitebox fuzzing[J]. *PLDI*, 2008, 43(6):206-215.

[16] The Explosion of the Ariane 5, *http://www-users.math.umn.edu/ arnold/disasters/ariane.html*.

[17] Wang T, Wei T, Gu G, et al. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection[C]. *S&P*, 2010:497-512.

[18] Pham V T, Böhme M, Roychoudhury A. Model-based whitebox fuzzing for program binaries[C]. *ASE*. 2016:543-553.

[19] Holler C, Herzig K, Zeller A. Fuzzing with Code Fragments[C]. *Usenix*, 2012:445–458.

[20] Dewey K, Roesch J, Hardekopf B. Language fuzzing using constraint logic programming[C]. *ASE*, 2014:725-730.

[21] Tassey, Gregory. The Economic Impacts of Inadequate Infrastructure for Software Testing[J]. *National Institute of Standards & Technology*, 2002, 15(3):125-125.

[22] Schwartz E J, Avgerinos T, Brumley D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)[C]. *S&P*, 2010:317-331.

[23] Cadar C, Dunbar D, Engler D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs[C]. *Usenix*, 2009:209-224.

[24] Woo M, Sang K C, Gottlieb S, et al. Scheduling black-box mutational fuzzing[C]. *CCS*, 2013:511-522.

[25] Allen D. Householder, Foote Jonathan M. Probability-Based Parameter Selection for Black-Box Fuzz Testing[J]. 2012.

[26] Godefroid P, Peleg H, Singh R. Learn&Fuzz: Machine learning for input fuzzing[C]. *ASE*, 2017:50-59.

[27] N. Nichols, M. Raugas, R. Jasper, and N. Hilliard, Faster fuzzing: Reinitialization with deep neural models, CoRR, vol. abs/1711.02807, 2017. [Online]. Available: *http://arxiv.org/abs/1711.02807*.

[28] Eddington, M. Peach fuzzer. *https://www.peach.tech/*.

[29] honggfuzz. *https://github.com/google/honggfuzz*.

[30] Maaten L, Hinton G. Visualizing data using t-SNE[J]. *Journal of machine learning research*, 2008, 9(Nov): 2579-2605.

[31] Gan S, Zhang C, Qin X, et al. CollAFL: Path Sensitive Fuzzing[C]. *S&P 2018.*

[32] SourceForge. *https://sourceforge.net/.*

[33] Zhai S, Cheng Y, Feris R, et al. Generative Adversarial Networks as Variational Training of Energy Based Models[J]. 2016.

[34] Chen X, Duan Y, Houthooft R, et al. InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets[C]. *NIPS*. 2016.

[35] Böhme M, Pham V T, Nguyen M D, et al. Directed greybox fuzzing[C]. *CCS,* 2017: 2329-2344.

[36] Corina J, Machiry A, Salls C, et al. DIFUZE: Interface Aware Fuzzing for Kernel Drivers[C]. *CCS*, 2017: 2123-2138.

[37] Xu W, Kashyap S, Min C, et al. Designing New Operating Primitives to Improve Fuzzing Performance[C]. *CCS*, 2017: 2313-2328.

[38] Han H S, Cha S K. IMF: Inferred Model-based Fuzzer[C]. *CCS*, 2017: 2345-2358.

[39] Peng H, Shoshitaishvili Y, Payer M. T-Fuzz: fuzzing by program transformation[C]. *S&P*, 2018.

[40] Chen P, Chen H. Angora: Efficient Fuzzing by Principled Search[C]. *S&P*, 2018.

[41] You W, Zong P, Chen K, et al. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits[C]. *CCS*, 2017: 2139-2154.

[42] Petsios T, Zhao J, Keromytis A D, et al. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities[C]. *CCS*, 2017: 2155-2168.

[43] Zhang H, Xu T, Li H, et al. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks[C]. *ICCV*. 2017: 5907-5915.

[44] Nguyen A, Yosinski J, Bengio Y, et al. Plug & play generative networks: Conditional iterative generation of images in latent space[J]. *arXiv preprint arXiv:*1612.00005, 2016.

[45] Gulrajani I, Ahmed F, Arjovsky M, et al. Improved training of wasserstein gans[C]. *NIPS*. 2017: 5769-5779.

[46] Liu M Y, Breuel T, Kautz J. Unsupervised image-to-image translation networks[C]. *NIPS*. 2017: 700-708.

[47] Zhu J Y, Zhang R, Pathak D, et al. Toward multimodal image-to-image translation[C]. *NIPS*. 2017: 465-476.

[48] Zhu J Y, Park T, Isola P, et al. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks[C]. *ICCV*. 2017:2242-2251.

[49] CVE. *http://cve.mitre.org/index.html.*

[50] AddressSanitizer. *http://clang.llvm.org/docs/AddressSanitizer.html.*

VII. NUMBER OF SEEDS VS. FUZZING PERFORMANCE

Now, we examine the relationship between the number of seed files and the fuzzing effectiveness. We use the same virtual machines with an Intel i7 CPU, 4.5GB memory and a Ubuntu 16.04 LTS system to run each fuzzing process for 72 hours. The results are shown in Table VIII, from which we can learn the following observations.

(1) Because mutation-based fuzzing tools such as AFL use byte flipping and so on to mutate files, the variation of the mutated files is slight and slow. Therefore, the fuzzing performance will be influenced if the number of seeds is too small due to the lack of seeds' diversity. Then, a proper number of seeds should be bigger than the lower bound. Since the performance of all the three fuzzing strategies are not obviously getting worse when the number of seeds in the initial set is 50, we come to a conclusion that 50 or more seeds is enough to bootstrap these fuzzing tools.

(2) In regard to the discovery of unique crashes, for `mpg321`, `SmartSeed` and `AFL-result` perform the best when the number of seeds is 50, while `random` discovers the most crashes when the seed number is 200. `SmartSeed` and `AFL-result` discover the most crashes on `sam2p` when the seed number is 100, while `random` finds the most unique crashes on `sam2p` with 300 seeds. For `mp42aac`, `SmartSeed` finds the most crashes when the seed number is 50. `random` and `AFL-result` discover the most crashes when the number of seeds is 300 and 100, respectively. Therefore, it seems that there is no obvious relationship between the number of discovered crashes and the number of seeds when there are enough seeds.

(3) As for the discovery of unique paths, for `mpg321`, `SmartSeed` finds the most paths when the number of seeds is 100, while `random` and `AFL-result` discover the most paths when they have 50 seeds. When fuzzing `sam2p`, `SmartSeed` finds the most paths with 200 seeds, while `random` uses 50 seeds to discover the most paths. `AFL-result` finds the most paths when the number of seeds is 100. For `mp42aac`, `SmartSeed`, `random` and `AFL-result` find the most paths when the number of seeds is 50, 200 and 100, respectively. It seems that if the number of the initial seeds is too big, the number of discovered paths will decrease rarely. We conjecture the reason is that the initial seeds detect more unique paths, and thus, the discovered paths later will be fewer. Therefore, it is a negligible relationship between the number of seeds in the initial set and the number of discovered paths. Note that it is hard to say 100 is a big number for the number of seeds in the initial set.

In summary, it seems 50 seeds in the initial seed set is enough to guide the fuzzing tools to detect crashes and paths, while there is no evident relationship between the number of seeds and the fuzzing performance. Therefore, it is OK to use 100 seeds as the initial seed set in the main body of our paper.

VIII. FURTHER ANALYSIS

A. Distribution

We would like to examine the distribution of the seeds generated by different strategies, and the distribution of the valuable files mutated from those seeds that trigger crashes.

To facilitate our analysis, we employ t-SNE [30], which is one of the best dimensionality reduction algorithms that can aggregate similar data together, to visualize the distribution. Specifically, we use `SmartSeed`, `random` and `AFL-result` to generate 300 seeds, respectively, and then visualize the distribution of these seeds in Fig. 6. Here, note that (1) the seeds of `AFL-result` are selected from the stored files of AFL that can trigger crashes or new paths, which are actually the mutated files of the `random`'s seeds using AFL as we described before (the default seed selection strategy of AFL is `random` in our implementation); and (2) we do not consider `peachset`, `hotset`, and `AFL-cmin`. This is mainly because similar to `random`, the seeds generated by these three schemes highly depend on the initial seed source dataset. It turns out that the seeds obtained by them exhibit similar distribution as `random`. Thus, we do not plot their distribution in Fig. 6 in order to make it more readable. From Fig. 6, we have the following observations.

(1) Although `AFL-result` is mutated from `random`, its distribution is far away from `random`. In other words, it takes time for `random` to discover valuable input files that trigger crashes or new paths. Compared with the distribution of `random`, the distribution of `SmartSeed` is closer to `AFL-result`. This implies that `SmartSeed` can discover crashes and paths faster than `random`.

(2) From Fig. 6, we learn that the distribution of `AFL-result` is more decentralized. Compared with `AFL-result`, the distribution of `SmartSeed` is more intensive. `SmartSeed` is also closer to the main part of `AFL-result`. These facts indicate that: when using `AFL-result` to fuzz an application, an input file may spend a longer time to be mutated into another valuable input file because of the discrete distribution. By contrast, `SmartSeed` may be mutated fast into the main part of valuable input files that can trigger crashes and paths of the objective application. Thus, `SmartSeed` is more effective compared with `AFL-result`.

Then, we leverage t-SNE to analyze the valuable mutated files of `SmartSeed + AFL`, `random + AFL` and `AFL-result + AFL`, i.e., the files are mutated from the seeds of `SmartSeed`, `random` and `AFL-result` that trigger unique crashes of objective applications. Since `SmartSeed` discovers more crashes than `random` and `AFL-result`, its points are more than those two. Note that the distribution of valuable files discovered by `SmartSeed` will only be more sparse if we use the same number of files to plot. The results are shown in Fig. 7, from which we have the following observations.

(1) The distribution of valuable files mutated from `AFL-result` and `random` are similar. Also, it seems difficult for the seeds of `AFL-result` and `random` to be mutated

TABLE VIII
UNIQUE CRASHES AND PATHS OF EACH OBJECTIVE APPLICATION WHEN USING DIFFERENT NUMBER OF FILES AS THE INITIAL SEED SET TO FUZZ.

	number	SmartSeed + AFL		random + AFL		AFL-result + AFL	
		unique crashes	unique paths	unique crashes	unique paths	unique crashes	unique paths
mpg321	50	**231**	1,029	48	**784**	**25**	**329**
	100	204	**1,060**	40	766	13	187
	200	164	1,051	**66**	565	10	126
	300	181	1,023	24	608	9	104
sam2p	50	37	1,358	23	**559**	21	404
	100	**50**	1,322	21	468	**36**	**719**
	200	49	**1,392**	27	458	30	372
	300	48	1,373	**28**	494	34	357
mp42aac	50	**176**	**1,226**	76	560	75	441
	100	118	658	80	329	**102**	**585**
	200	108	879	83	**626**	31	340
	300	106	502	**91**	520	53	473
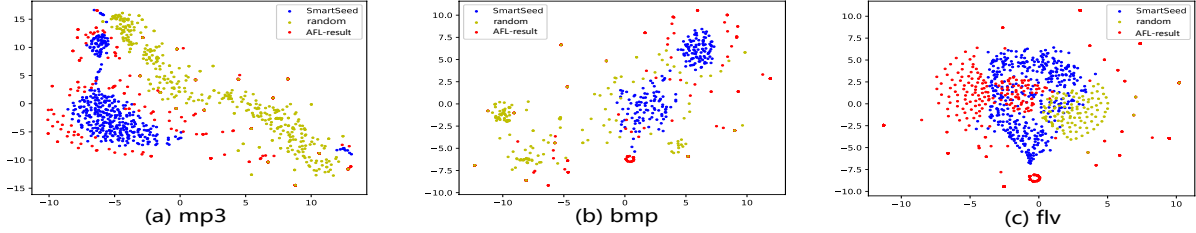
Fig. 6. Similarity distribution of the seeds of `SmartSeed`, `random` and `AFL-result`.
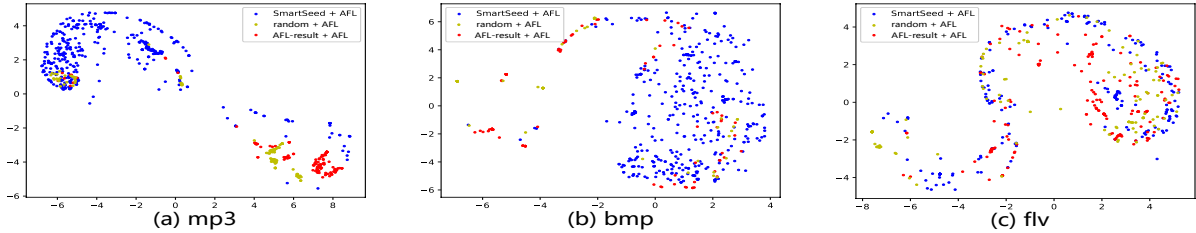
Fig. 7. Similarity distribution of valuable mutated input files of `SmartSeed`, `random` and `AFL-result` that triggered crashes.

into the distant points that can trigger crashes, which then limits their fuzzing performance.

(2) On the contrary, the points of the valuable mutated files of `SmartSeed` spread in a larger area. This demonstrates that the seed files generated by `SmartSeed` are easier to be mutated into the discrete valuable input files that can trigger crashes.

(3) We can learn from Fig. 6 and Fig. 7 that although the distributions of `AFL-result` and `random` are more discrete than `SmartSeed`, the valuable input files discovered by `SmartSeed`, which trigger unique crashes of objective applications, cover a larger area. In other words, the relatively-concentrate distribution of the seeds generated by `SmartSeed` does not limit them to be mutated into discrete input files and trigger more unique crashes.

In summary, the distribution of `SmartSeed` is dense and closer to the main part of `AFL-result`. Meanwhile, the seeds of `SmartSeed` seem easier to be mutated into valuable input files that can trigger crashes. These observations may explain the better performance of `SmartSeed` from another angle.

IX. LIMITATIONS AND FUTURE WORK.

As the focus of this paper, `SmartSeed` in its current form is designed for genetic algorithm based fuzzing. Therefore, it is not suitable to use `SmartSeed` for generating highly-structured input files. We take this issue as the future work of this paper and keep extending our system. From the performance perspective, `SmartSeed` could be improved from many aspects. For instance, a better generative model could be designed. Also, it is interesting to study the best working scope of different generative models and how to further improve the model training process. From the evaluation perspective, our primary goal in this paper is to demonstrate the performance and usability of `SmartSeed`. Certainly, more evaluations can be conducted to comprehensively evaluate `SmartSeed`, e.g., considering more applications and more formats.

X. Related Work

In this section, we briefly introduce the related work.

Mutation-based Fuzzing. As a representative of mutation-based fuzzing, AFL [11] employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover the valuable input files that trigger new paths or unique crashes. Because of its high-efficiency and ease of use, AFL is one of the most popular fuzzing tools. Based on AFL, Böhme et al. presented a fuzzing tool named *AFLFast* that can detect more paths by prioritizing low-frequency paths [2]. In [35], Böhme et al. combined a simulated annealing-based power schedule scheme with AFL and presented *AFLGo*. Xu et al. implemented three new operating primitives specialized for improving the fuzzing performance of large-scale tasks on multi-core machines [37]. In [31], Gan et al. presented a solution to mitigate path collisions, which can be combined with AFL and AFLFast to construct CollAFL and CollAFL-fast, respectively.

Many other works focus on combining fuzzing with other bug detection technologies such as taint tracing, symbolic execution and program analysis. Wang et al. combined fuzzing with dynamic taint analysis and symbolic execution techniques and presented a fuzzing tool named *TaintScope* [17]. Haller et al. presented a fuzzing tool named *Dowser*, which takes taint tracking, program analysis and symbolic execution into consideration [7]. Sang et al. considered to employ white-box symbolic analysis in their fuzzing tool design [4]. Stephens et al. also considered to involve selective symbolic execution into their fuzzing tool named *Driller* [5]. Pham et al. combined input model-based approaches with symbolic execution and presented Model-based Whitebox Fuzzing (MoWF) [18].

To improve the coverage, Rawat et al. presented an application-aware evolutionary fuzzing tool named *VUzzer* [3]. *VUzzer* uses static and dynamic analysis to analyze the priority of paths. In [39], Peng et al. presented *T-Fuzz*, which uses a dynamic tracing based technique to detect and remove the checks in objective applications to improve the code coverage. By using scalable byte-level taint tracking, context-sensitive branch count, gradient descent based search, shape and type inference and input length exploration to solve path constraints, Angora presented by Chen et al. can increase the branch coverage of objective applications [40].

Note that SmartSeed is designed as a generic system to generate valuable seed files for and to be easily compatible with mutation-based fuzzing. With the seeds generated by SmartSeed, we expect to improve the performance of mutation-based fuzzing.

Generation-based Fuzzing. Generation-based fuzzing tools are designed to generate input files with specific input formats. Following this track, Godefroid et al. used the grammar-based specification of valid highly-structured input files to improve the performance of fuzzing [15]. Holler et al. presented *Lang-Fuzz* to fuzz the applications with highly-structured inputs such as JavaScript interpreters [19]. In order to fuzz compilers and interpreters, Dewey et al. proposed to use Constraint Logic Programming (CLP) for the program generation [20]. Recently, Wang et al. presented a novel data-driven seed generation approach named *Skyfire* [6], which uses Probabilistic Context-Sensitive Grammar (PCSG) to learn the syntax features and semantic rules from the training set. In [26], Godefroid et.al. presented a RNN-based machine-learning technique to generate highly-structured format files such as PDF. The method not only can pass the format checks with a high probability, but also can improve the code coverage.

The primary difference between SmartSeed and generation-based fuzzing approaches is that our method is used to generate binary seed files such as image, music and video, while they focus on improving the fuzzing efficiency of applications with highly-structured input formats.

Other Fuzzing Strategies. As for kernel vulnerabilities, Corina et al. presented an interface-aware fuzzing tool named *DIFUZE* to automatically generate inputs for kernel drivers [36]. Han et al. proposed a novel method called *model-based API fuzzing* and presented *IMF* for testing commodity OS kernels [38]. You et al. presented a novel technique named *SemFuzz* [41] that can learn from vulnerability-related texts such as CVE reports and automatically generate Proof-of-Concept (PoC) exploits. Petsios et al. focused on algorithmic complexity vulnerabilities and proposed *SlowFuzz* to generate inputs to trigger the maximal resource utilization behavior of applications and algorithms [42].

Seed Selection. To figure out how to select a better initial seed set, Allen and Foote presented an algorithm to consider the parameter selection and automated selection of seed files [25]. Woo et al. developed a framework to evaluate 26 randomized online scheduling algorithms to see which can schedule better seeds to fuzz a program [24]. Rebert et al. evaluated six seed selection strategies of fuzzing and presented several interesting conclusions about seed sets [1]. They also showed the necessity to select a good seed set to improve the efficiency of fuzzing. Recently, Nichols et.al. showed that using the generated files of GAN to reinitialize AFL can potentially find more unique paths of ethkey [27]. However, they neither described the model in detail nor provided any prototype. Thus, we cannot reproduce their model and compare it with SmartSeed.

Different from existing research, we focus on designing a generic seed generation system leveraging start-of-the-art machine learning techniques. We also demonstrate its effectiveness, robustness and compatibility through extensive experiments.

GAN Models. In 2014, Goodfellow et al. presented a new unsupervised learning framework named Generative Adversarial Networks (GAN) [8]. To improve GAN, Radford et al. tried multiple combinations of machine learning models to construct better generative and discriminative models for GAN, and presented the Deep Convolutional GAN (DCGAN) model [9]. Later, Zhai et al. combined GAN with an Energy Based Model (EBM) and proposed VGAN that works by minimizing a variational lower bound of the negative log likelihood of EBM [33]. Chen et al. combined GAN with mutual information and presented InfoGAN [34], which can unsupervisedly learn interpretable representations. Recently, Arjovsky et al. presented the Wasserstein GAN (WGAN) model by using the approximation of the Wasserstein distance

as the loss function [10].

Since GAN and its variants can use unsupervised learning methods to generated more realistic data, they have been applied in many applications such as high-quality image generation [43], [44], [45] and image translation [46], [47], [48]. In this paper, we extend the application of GAN to improve the performance of fuzzing.

REFERENCES

[1] Rebert A, Sang K C, Grieco G, et al. Optimizing seed selection for fuzzing[C]. *Usenix*, 2014:861-87

[2] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain[C]. *CCS*, 2016, PP(99):1-1.

[3] Rawat S, Jain V, Kumar A, et al. VUzzer: Application-aware Evolutionary Fuzzing[C] *NDSS*. 2017.

[4] Sang K C, Woo M, Brumley D. Program-Adaptive Mutational Fuzzing[C] *S&P*, 2015:725-741.

[5] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution[C] *NDSS*. 2016.

[6] Wang J, Chen B, Wei L, et al. Skyfire: Data-Driven Seed Generation for Fuzzing[C] *S&P*, 2017:579-594.

[7] Haller I, Slowinska A, Neugschwandtner M, et al. Dowsing for overflows: a guided fuzzer to find buffer boundary violations[C]. *Usenix*. 2013:49-64.

[8] Goodfellow I J, Pouget-Abadie J, Mirza M, et al. Generative adversarial nets[C] *NIPS*. 2014:2672-2680.

[9] Radford A, Metz L, Chintala S. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks[J]. *Computer Science*, 2015.

[10] Arjovsky M, Chintala S, Bottou L. Wasserstein GAN. 2017.

[11] American Fuzzy Lop (AFL), *http://lcamtuf.coredump.cx/afl/*.

[12] Therac-25 accidents, *https://en.wikipedia.org/wiki/Therac-25*.

[13] Clause J, Li W, Orso A. Dytan: a generic dynamic taint analysis framework[C].*ISSTA*. 2007:196-206.

[14] Kang M G, Mccamant S, Poosankam P, et al. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation[C]. *NDSS*. 2011.

[15] Godefroid P, Kieżun A, Levin M Y. Grammar-based whitebox fuzzing[J]. *PLDI*, 2008, 43(6):206-215.

[16] The Explosion of the Ariane 5, *http://www-users.math.umn.edu/ arnold/disasters/ariane.html*.

[17] Wang T, Wei T, Gu G, et al. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection[C]. *S&P*, 2010:497-512.

[18] Pham V T, Böhme M, Roychoudhury A. Model-based whitebox fuzzing for program binaries[C]. *ASE*. 2016:543-553.

[19] Holler C, Herzig K, Zeller A. Fuzzing with Code Fragments[C]. *Usenix*, 2012:445–458.

[20] Dewey K, Roesch J, Hardekopf B. Language fuzzing using constraint logic programming[C]. *ASE*, 2014:725-730.

[21] Tassey, Gregory. The Economic Impacts of Inadequate Infrastructure for Software Testing[J]. *National Institute of Standards & Technology,* 2002, 15(3):125-125.

[22] Schwartz E J, Avgerinos T, Brumley D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)[C]. *S&P,* 2010:317-331.

[23] Cadar C, Dunbar D, Engler D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs[C]. *Usenix,* 2009:209-224.

[24] Woo M, Sang K C, Gottlieb S, et al. Scheduling black-box mutational fuzzing[C]. *CCS*, 2013:511-522.

[25] Allen D. Householder, Foote Jonathan M. Probability-Based Parameter Selection for Black-Box Fuzz Testing[J]. 2012.

[26] Godefroid P, Peleg H, Singh R. Learn&Fuzz: Machine learning for input fuzzing[C]. *ASE*, 2017:50-59.

[27] N. Nichols, M. Raugas, R. Jasper, and N. Hilliard, Faster fuzzing: Reinitialization with deep neural models, CoRR, vol. abs/1711.02807, 2017. [Online]. Available: *http://arxiv.org/abs/1711.02807*.

[28] Eddington, M. Peach fuzzer. *https://www.peach.tech/*.

[29] honggfuzz. *https://github.com/google/honggfuzz*.

[30] Maaten L, Hinton G. Visualizing data using t-SNE[J]. *Journal of machine learning research*, 2008, 9(Nov): 2579-2605.

[31] Gan S, Zhang C, Qin X, et al. CollAFL: Path Sensitive Fuzzing[C]. *S&P 2018*.

[32] SourceForge. *https://sourceforge.net/*.

[33] Zhai S, Cheng Y, Feris R, et al. Generative Adversarial Networks as Variational Training of Energy Based Models[J]. 2016.

[34] Chen X, Duan Y, Houthooft R, et al. InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets[C]. *NIPS*. 2016.

[35] Böhme M, Pham V T, Nguyen M D, et al. Directed greybox fuzzing[C]. *CCS,* 2017: 2329-2344.

[36] Corina J, Machiry A, Salls C, et al. DIFUZE: Interface Aware Fuzzing for Kernel Drivers[C]. *CCS*, 2017: 2123-2138.

[37] Xu W, Kashyap S, Min C, et al. Designing New Operating Primitives to Improve Fuzzing Performance[C]. *CCS*, 2017: 2313-2328.

[38] Han H S, Cha S K. IMF: Inferred Model-based Fuzzer[C]. *CCS*, 2017: 2345-2358.

[39] Peng H, Shoshitaishvili Y, Payer M. T-Fuzz: fuzzing by program transformation[C]. *S&P*, 2018.

[40] Chen P, Chen H. Angora: Efficient Fuzzing by Principled Search[C]. *S&P*, 2018.

[41] You W, Zong P, Chen K, et al. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits[C]. *CCS*, 2017: 2139-2154.

[42] Petsios T, Zhao J, Keromytis A D, et al. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities[C]. *CCS,* 2017: 2155-2168.

[43] Zhang H, Xu T, Li H, et al. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks[C]. *ICCV*. 2017: 5907-5915.

[44] Nguyen A, Yosinski J, Bengio Y, et al. Plug & play generative networks: Conditional iterative generation of images in latent space[J]. *arXiv preprint arXiv:*1612.00005, 2016.

[45] Gulrajani I, Ahmed F, Arjovsky M, et al. Improved training of wasserstein gans[C]. *NIPS*. 2017: 5769-5779.

[46] Liu M Y, Breuel T, Kautz J. Unsupervised image-to-image translation networks[C]. *NIPS*. 2017: 700-708.

[47] Zhu J Y, Zhang R, Pathak D, et al. Toward multimodal image-to-image translation[C]. *NIPS*. 2017: 465-476.

[48] Zhu J Y, Park T, Isola P, et al. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks[C]. *ICCV*. 2017:2242-2251.

[49] CVE. *http://cve.mitre.org/index.html*.

[50] AddressSanitizer. *http://clang.llvm.org/docs/AddressSanitizer.html*.