

# DeFuzz: Deep Learning Guided Directed Fuzzing

1<sup>st</sup> Xiaogang Zhu

*Swinburne University Of Technology*  
Melbourne, VIC, Australia  
xiaogangzhu@swin.edu.au

2<sup>nd</sup> Shigang Liu

*Swinburne University Of Technology*  
Melbourne, VIC, Australia  
shigangliu@swin.edu.au

3<sup>rd</sup> Xian Li

*Swinburne University Of Technology*  
Melbourne, VIC, Australia  
xianli1990@outlook.com

4<sup>th</sup> Sheng Wen

*Swinburne University Of Technology*  
Melbourne, VIC, Australia  
swen@swin.edu.au

5<sup>th</sup> Jun Zhang

*Swinburne University Of Technology*  
Melbourne, VIC, Australia  
junzhang@swin.edu.au

6<sup>th</sup> Camtepe Seyit

*CSIRO & DATA61*  
Marsfield, NSW, Australia  
seyit.camtepe@data61.csiro.au

6<sup>th</sup> Yang Xiang

*Swinburne University Of Technology*  
Melbourne, VIC, Australia  
yxian@swin.edu.au

**Abstract**—Fuzzing is one of the most effective technique to identify potential software vulnerabilities. Most of the fuzzers aim to improve the code coverage, and there is lack of directedness (e.g., fuzz the specified path in a software). In this paper, we proposed a deep learning (DL) guided directed fuzzing for software vulnerability detection, named DeFuzz. DeFuzz includes two main schemes: (1) we employ a pre-trained DL prediction model to identify the potentially vulnerable functions and the locations (*i.e.*, vulnerable addresses). Precisely, we employ Bidirectional-LSTM (BiLSTM) to identify attention words, and the vulnerabilities are associated with these attention words in functions. (2) then we employ directly fuzzing to fuzz the potential vulnerabilities by generating inputs that tend to arrive the predicted locations. To evaluate the effectiveness and practical of the proposed DeFuzz technique, we have conducted experiments on real-world data sets. Experimental results show that our DeFuzz can discover coverage more and faster than AFL. Moreover, DeFuzz exposes 43 more bugs than AFL on real-world applications.

**Index Terms**—fuzz testing, deep learning, software security, vulnerability, static analysis, dynamic analysis

## I. INTRODUCTION

Computer software is a crucial part of the modern world. The software makes the world become smart than before. However, cyber threats attributed to vulnerabilities in software are becoming a serious security problem [1]. For example, it is estimated that it will cost \$6 trillion to combat cybercrime by 2021, which is double of the cost in 2015. Therefore, the vulnerabilities in the software should be ideally identified and fixed before the software get deployed.

Many algorithms have been developed for software vulnerability detection [2], [3], [4]. Among these techniques, fuzzing is one of the most effective approach to identify potential software vulnerabilities [5]. However, fuzzing has the problem of time-consuming because it identify bugs using randomly generated inputs. Fuzz testing usually hard to arrive all the code coverage given a real-world scenario. For example, it is very hard to generate inputs for some sanity checks with

the increasing size of the real-world programs [6]. Moreover, previous study showed that vulnerable functions usually come from a small part of the whole program. To address these problems, directed greybox fuzzing, which uses the inputs that are generated with the objective of reaching specific locations, has been developed. Nevertheless, greybox fuzzing usually has the problem that it cannot be effectively directed [7]. Therefore, there is still necessary to research into the directed fuzzing, which can fuzz the specific targeted locations with less of resources.

In this work, we take the advantages of deep learning [8], and developed a deep learning directed fuzzing for bugs identification. Deep learning has shown successful in software vulnerability detection [9], [10], specifically in learning the high-level representations for both vulnerable and not-vulnerable programming features. The high-level representations usually contains richer useful information than the generic hand-crafted features [11]. However, deep learning-based techniques usually has high false positives (e.g., more than 20%) [11], [9], [10]. In this work, we employ deep learning to identify the potential vulnerable functions and the vulnerability locations. Then we run the directed fuzzing to fuzz the vulnerable locations to reduce the false positive rate.

Specifically, given a software program, we first extract functions from the program, and then we extract Abstract Syntax Trees (ASTs) for each function. The ASTs we collected will be used as the training samples. The training samples are from LibTIFF, LibPNG, and FFmpeg, which are three open-source projects. We label the vulnerable functions by following the Common Vulnerability and Exposures and the National Vulnerability Database. The assumption is that if there are at least one vulnerability in the function, then the function will be labeled as vulnerable, otherwise, it will be labeled as not vulnerable. Then, we train an attention-based deep learning model using the ground truth. The pre-trained model will be applied to the test cases (*i.e.*, a piece of program

or a software program). The outputs will be the locations of the potential vulnerabilities (the vulnerabilities are associated with these attention words in functions). Afterwards, we fuzz the vulnerable locations using directed fuzzing [12]. We call this as deep learning guided directed fuzzing, namely DeFuzz. The main contributions of this work are as follows:

- We propose to employ attention-based deep learning to highlight code keyword, which will be used to identify the penitential vulnerable locations (i.e., vulnerable addresses) for real-world security problems.
- We propose to use deep learning to guide the directed fuzzing. When deep learning can successfully predict the exact or the neighbours of vulnerability locations, directed fuzzing is efficient to detect vulnerabilities.
- We conducted experiments on real-world scenarios. Experimental results show that the proposed DeFuzz can discover more coverages and faster than AFL. For example, DeFuzz exposes 43 more bugs than AFL on real-world applications.

## II. RELATED WORK

In this section, we briefly review of the related work from two perspectives: machine learning-based vulnerability detection and fuzz testing for software vulnerability discovery. For more information, please refer to [2], [3], [4], [13], [14].

### A. Machine Learning based vulnerability detection

Machine learning has been widely used for software vulnerability detection. Morrison et al. [36] investigate the machine learning techniques for software vulnerability identification, their study show that machine learning techniques is useful in vulnerability detection. However, the classification performance based on source code usually performs better than binary code level.

Machine learning techniques for software vulnerability detection can be divided into two stages: feature learning, and model building. Feature learning is a case by case process. For example, Shar and Tan [15] propose to consider static code attributes for model building to predict specific program statements for SQL injection and cross site scripting. Yamaguchi et al. [16] propose to use clustering algorithms on code property graph based on C source code for taint-style vulnerability discovery. Alves et al. [17] have done an experimental study over machine learning techniques, their experimental results show that random forest usually has a better performance. DisovRE [18] uses control flow graph for function similarity match to identify potential bugs given a piece of software. VDiscover [19] employs lightweight static and dynamic features to identify potential vulnerabilities given a test case. SemHunt [20] proposes a scheme to predict vulnerable functions and their pairs of patched and unpatched functions using the binary executable programs. Calzavara et al. [21] propose to consider HTTP requests as features and employ supervised machine learning to detect Cross-Site Request Forgery vulnerabilities.

Meanwhile, deep learning has been widely accepted and applied for software vulnerability detection. This is because deep learning has the capability of learning high-level feature representations given a program. Lin et al. [22] propose to employ LSTM for function-level vulnerability detection based on ASTs. Li et al. [9] consider BiLSTM using code gadgets as features for software vulnerability detection. Liu et al. [1] realize that there is class imbalance problem [23] in software vulnerability detection, and propose a fuzzy-based oversampling algorithm using ASTs for vulnerability detection. Furthermore, a deep learning based cross domain software detection has been developed [24], this study employs metric transfer learning framework to minimize the distribution differences between the source domain and target domain. Liu et al. [10] propose to employ attention-based deep learning using binary instruction as features for binary-level vulnerability detection. For more information about software vulnerability detection using machine, please refer to [25].

### B. Directed Fuzzing

Coverage-guided fuzzing is a widely-used fuzzing solution due to its effectiveness [26]–[35]. Coverage-guided fuzzing aims to cover as much coverage as possible. The assumption is that with more coverage discovered, it has higher chance to expose bugs. Driller [34] uses symbolic execution to solve path constraints, which is the major road rock to discover more coverage, for fuzzing. CollAFL [35] improves the coverage via resolving edge collision. Angora [36] and Matryoshka [29] regard a single path constraint as a black-box function, and use gradient descent to bypass the path constraint. As some path constraints compare variables with constant values, REDQUEEN [31] uses the constant values to bypass constraints. The mutation of the entire input for a program requires much time. Therefore, GREYONE [27] first finds the relation between input bytes and path constraints, and then only mutates the related bytes. This solution largely reduce the searching space of input. As the random nature of fuzzing, the execution speed is critical for fuzzing. Full-speed fuzzing [28] improves the execution speed of fuzzing via only tracing execution that discovers new coverage. Although coverage-guided fuzzing has achieved a great success, it has problems to detect vulnerabilities efficiently. The reason is that, coverage-guided fuzzing spends almost the time for each path, wasting time for the non-vulnerability paths.

To improve the efficiency of coverage-guided fuzzing, directed fuzzing aims to cover the potential bug locations, which is more effective than coverage-guided fuzzing. Directed fuzzing is one of the most promising fuzzing solutions to detect vulnerabilities. Directed fuzzing instruments some potential bug locations and guides fuzzing to reach the locations. AFLGo [12] utilises the commit information from Github to set bug locations. Then, it assigns more mutation time for the execution paths that are closer to these locations. SAVIOR [37] sets the potential bug locations using Undefined Behavior Sanitizer (UBSan). UBSan can detect variables that are used before definition. Besides UBSan, other sanitizers can

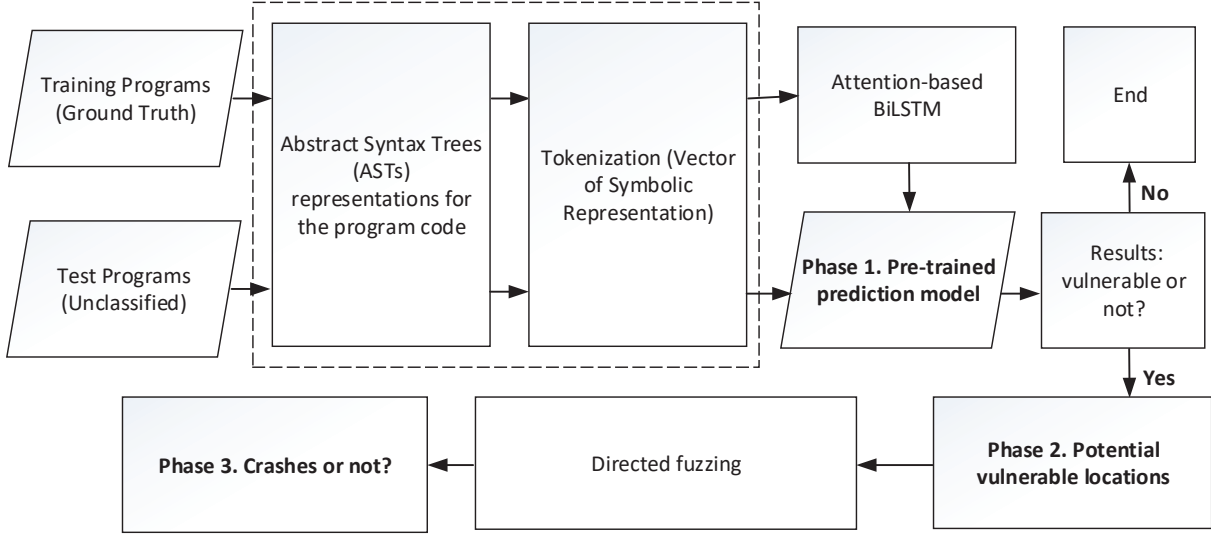


Fig. 1. Framework of the proposed DeFuzz. There are three phases: prediction model training, potential vulnerable location identification, and directed fuzzing.

```

375 case PUSH_CONSTANT16: /* CONSTANT16 */
376     if (act->p.Constant16 > poolcounter)
377     {
378         SWF_warn("WARNING: retrieving constants not present in the pool.\n");
379         break;
380     }
381     t=malloc(strlen(pool[act->p.Constant16])+3); /* 2 '\0's and a NULL */
382     strcpy(t, "");
383     strcat(t, pool[act->p.Constant16]);
384     strcat(t, "");
385     return t;

```

Fig. 2. Location of potential vulnerability identified from a piece of code of (or subroutine of) getString() function from decompile.c file.

also be used to set the potential bug locations. ParmeSan [38] regards the sanitization instrumentation as the target locations and guides fuzzing to these interesting basic blocks. To discover memory corruption vulnerabilities, TortoiseFuzz [39] statically analyses locations of sensitive memory operations, and then guides fuzzing to these memory related locations. Memory corruption is a common vulnerability among all kinds of vulnerabilities. However, in this paper, we utilise the results from deep learning to set the potential bug locations.

### III. PROPOSED DEFUZZ

In this section, we discuss the main idea of DeFuzz. DeFuzz is a deep learning guided directed fuzzing for software vulnerability identification. Fig.1 is the framework of the proposed DeFuzz. As shown in Fig.1, one can see there are three phases in the proposed scheme: 1) Prediction model training; 2) potential vulnerable location identification; 3) fuzz the potential vulnerable locations using directed fuzzing.

In the first phases, we train the attention-based deep learning model based on the ground truth we collected. This model is the pre-trained prediction model as we can see from Fig.1. In this study, the ground truth data samples are C/C++ functions<sup>1</sup> with labels which means the samples will be labeled as either vulnerable or not vulnerable [24], [22]. The data samples are

TABLE I  
CRASHING TRACE OF LISTACTION\_D. FUZZING HAS TO GENERATE AN INPUT THAT EXERCISES ALL THE TEN LISTED FUNCTIONS.

Functions in a Crashing Trace	File & Line
main	main.c:354
readMovie	main.c:281
outputBlock	outputtxt.c:2933
outputSWF_DOACTION	outputtxt.c:1620
decompile5Action	compile.c:3517
decompileActions	decompile.c:3494
decompileAction	decompile.c:3413
decompileSingleArgBuiltInFunctionCall	decompile.c:2994
newVar_N	decompile.c:725
getString	decompile.c:381

from three open-source projects including FFmpeg, LibPNG, and LibTIFF<sup>2</sup>. We collect the sources from Github, and then we manually labeled 417 vulnerable functions and 6,860 not-vulnerable functions. The assumption is that if there is at least one vulnerability identified in the function, then the function will be labelled as vulnerable, otherwise, it will be labeled as not vulnerable. Then, we extract ASTs representations for the program code at function level (a detailed description of ASTs can be found in [1], [11]). The ASTs are tokenized into AST sequences before they are fed to the the attention-based Bidirectional LSTM (BiLSTM) for feature representation learning. Finally, the fully connected neural network will be employed to train the prediction model based on the feature vector representations.

The second phase is used to identify potential vulnerabilities given any test cases. The inputs are C or C++ programs. The outputs are the vulnerable locations at function level (i.e., the test cases are programs at function level, the results will be the locations of potential vulnerabilities regarding each function). It is worth to note that the location of the vulnerabilities is identified by using the attention words pro-

<sup>1</sup>Function means a group of statements that together perform a task.

<sup>2</sup>Download from: <https://github.com/cybercodeintelligence/CyberCI>.

duced by the attention-based BiLSTM model. We consider attention-based deep learning to train the prediction model because the attention words produced by the model can be helpful for us to identify the location of the vulnerabilities. Fig.2 gives an example, we extracted *getString()* from *decompile.c* file of the *libming* project as a test case for the pre-trained model. When we fed this function to our pre-trained model for testing. The output of the attention words are: *malloc*, *Constant16*, *strlenext* and *pool*. After we manually check with *getString()* function. We then identified the potential vulnerability is from the line 381 of  $t = \text{malloc}(\text{strlenext}(\text{pool}[\text{act} - > p.\text{deconstant16}]) + 3)$ .

In the third phase, we employ the directed fuzzing to fuzz the vulnerable locations identified in the second phase. Specifically, we use AFLGo as the directed fuzzing. When AFLGo gets the potential vulnerability locations, which are generated by deep learning, AFLGo calculates distance between the current execution path and the target locations (Details can be found in [12]). Then, AFLGo assigns mutation chances based on the distance, *i.e.*, the smaller the distance is, the more mutation chances the path is assigned. Therefore, directed fuzzing spends more time on execution paths that examine target locations, which improves the efficiency of detecting vulnerabilities. Note that, although we use AFLGo to demonstrate the efficacy of our DeFuzz, we will use other directed fuzzers or design our own one in the future. As an example, Table I shows the bug location in the 381st line of the file *decompile.c* in program *libming*, and the bug is exposed running application *listaction\_d*. As shown in Table V (refer to Section IV), deep learning precisely predicts this bug location, and fuzzing is guided to this bug location. Starting from the function *main()*, fuzzing has to generate an input that exercises another nine functions and trigger the bug in function *getString()*. The 381st code line of function *getString()* calls *malloc()*, which causes a segmentation fault.

#### IV. EXPERIMENTS

We evaluate our DeFuzz on eight real-world open-sourced programs, which are shown in Table II. The eight target programs include seven programs from *libming* [40] library, which deals with *.swf* files, and *gif sponge* from *gif*, which manipulates GIF images. The parameters used for the eight programs are shown in the fifth column of Table II. These applications are widely used in the real world, and their security will have a significant impact on the applications developed based on them. Therefore, we chose them as our target programs. We compare our DeFuzz to AFL with the same configure, *i.e.*, the same initial input and timeout. We set the timeout as 24 hours, *i.e.*, the execution will terminate when fuzzing runs more than 24 hours. We run the experiments on the computer with *AMD Ryzen Threadripper 2990WX 32-core Processor 128GB RAM*. Each program runs on a single core. We will evaluate the performance of our DeFuzz in terms of execution speed, code coverage, and bug discovery.

TABLE II  
THE NUMBER OF UNIQUE CRASHES. DEFUZZ DETECTS MORE CRASHES THAN AFL.

Application	Version	DeFuzz	AFL	Parameters
listaction_d	commit 50098	10	16	@ @
listswf_d		162	216	
swftocxx		13	0	
swftoperl		87	0	
swftophp		166	0	
swftopython		155	0	
swftotcl		165	0	
gif sponge	commit 72e1f	11	0	

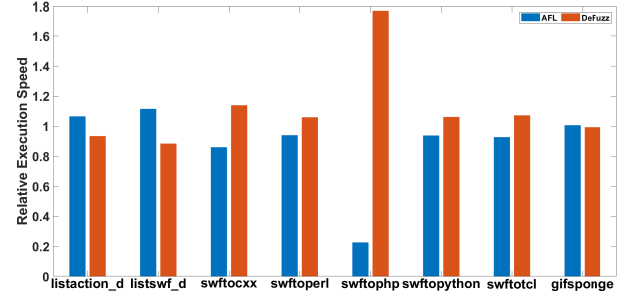


Fig. 3. Relative Execution Speed. The relative execution speed is the execution speed of each fuzzer compared to the average execution speed. A higher relative execution speed indicates more test cases being examined in the same time. The execution speed of DeFuzz is close to AFL's.

#### A. Execution Speed

The execution speed has a significant influence on fuzzing due to the random nature of fuzzing. With a higher execution speed, fuzzing saves time to run more executions. Fig.3 shows the relative execution speed of each application. For each fuzzer, we first calculate the average execution speed of each application. Then, the relative execution speed is the value that average execution speed of an individual fuzzer is divided by the average execution speed of both fuzzers.

As shown in Fig.3, the relative execution speed of AFL is faster than DeFuzz on two applications, *Listaction\_d*

TABLE III  
THE NUMBER OF REAL-WORLD BUGS AND TYPES. DEFUZZ DISCOVERS MORE BUGS THAN AFL.

Applications	DeFuzz	AFL
Detected Bugs per Project		
listaction_d	5	6
listswf_d	12	14
swftocxx	6	0
swftoperl	8	0
swftophp	10	0
swftopython	8	0
swftotcl	12	0
giflib	2	0
Detected bugs Type		
out-of-memory	✓	✓
double free	✓	×
Total	63	20

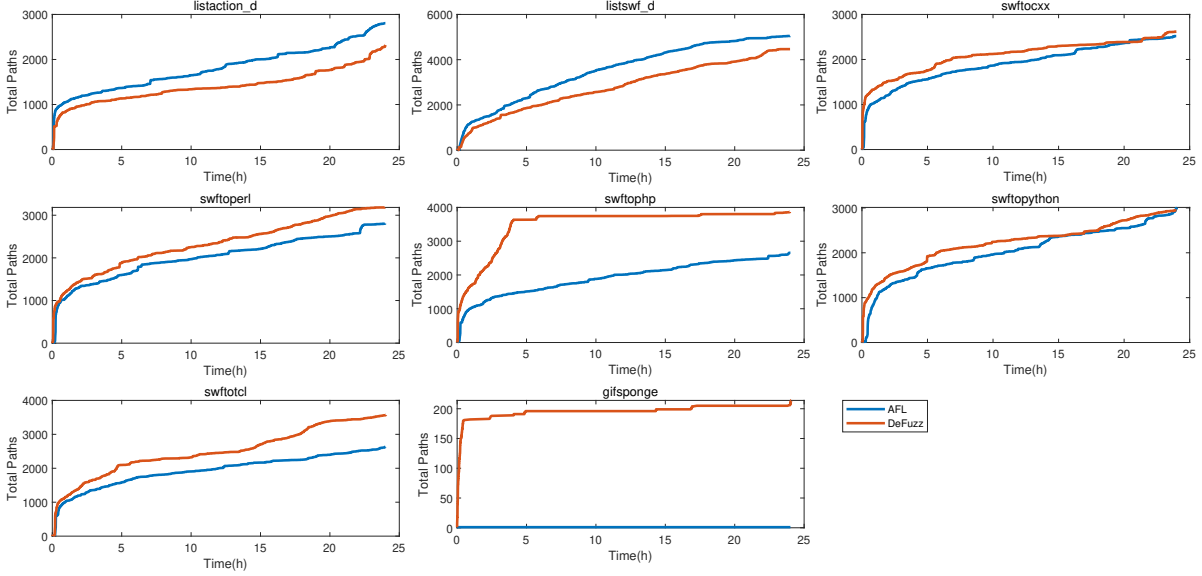


Fig. 4. The number of paths. On average, DeFuzz discovers more paths than AFL.

TABLE IV  
THE TIME OF DETECTING FIRST CRASHES BY AFL. OVER 24 HOURS,  
AFL DETECTS THE FIRST CRASHES.

Application	Time (h)
swiftocxx	26.8
swftoperl	25.7
swftophp	27.4
swftopython	25.5
swftotcl	26.8

and Listswf\_d. On the other hand, DeFuzz runs programs close or faster than AFL on the other six applications. On the application gifspoon, although the execution speeds of the two fuzzers are close, AFL discovers only one path during the experiments and exposes zero crashes. On the other hand, DeFuzz discovers much more crashes than AFL. On average, the execution speed of AFL and DeFuzz is close to each other, and DeFuzz is slightly faster than AFL. Excluding swftophp, DeFuzz has a relative execution speed 4% higher than AFL.

For coverage-guided fuzzing, such as AFL, it aims to discover as much coverage as possible. The assumption is that with a higher coverage, it has a higher chance to expose vulnerabilities. However, this might not always be true. Therefore, directed fuzzing, such as AFLGo, prefers to explore paths that are closer to the potential vulnerability locations. As DeFuzz is a directed fuzzer, it sets potential vulnerable locations utilising Deep Learning.

Fig.4 shows that DeFuzz discovers more coverage than AFL on six applications except for listaction\_d and listswf\_d. Although DeFuzz discovers less coverage than AFL on applications listaction\_d and listswf\_d, DeFuzz discovers more crashes than AFL, as shown in Fig.5. Excluding gifspoon, the overall code coverage of DeFuzz

is only 4% more than that of AFL, but DeFuzz detects three times as many crashes as AFL. Therefore, while DeFuzz is a directed fuzzing, it still discovers more coverage than AFL.

### B. Bug Discovery

In this section, we will evaluate the performance of DeFuzz in terms of crash discovery and bug discovery. The number of unique crashes is an important indicator of the effectiveness of fuzzing. The more crashes are detected, the greater probability of finding vulnerabilities. TABLE II shows that, within 24 hours, DeFuzz has successfully detected crashes on every target program. On the other hand, AFL did not detect any crash on the same applications except for listaction\_d and listswf\_d. Therefore, DeFuzz is more effective than AFL to expose crashes.

We de-duplicated the crashes utilising afl-collect [41], and then verified bugs manually based on the crash stacks reproduced by GDB [42]. The number of bugs discovered by fuzzers is shown in TABLE III. Overall, DeFuzz discovers much more bugs than AFL on six applications. Meanwhile, the two fuzzers expose close number of bugs on applications listaction\_d and listswf\_d. Note that we ran each trial for 24 hours, and AFL cannot expose crashes within 24 hours on six applications. We continued the experiments for AFL until it exposed crashes on the six applications that could not expose crashes within 24 hours. Table IV shows the time that AFL exposes the first crash on five applications. We could not find crashes on application gifspoon. The bug locations of each application are shown in TABLE V, where the file name is the file in which bug happens and the line number is the corresponding line in the file. The bug locations predicted by deep learning are also shown in TABLE V.

Comparing the real bug locations with the predicted ones, TABLE V shows that deep learning has successfully predicted

TABLE V

THE PREDICTED BUG LOCATIONS OF DEEP LEARNING AND THE REAL BUG LOCATIONS. MOST REAL BUG LOCATIONS ARE SUCCESSFULLY PREDICTED BY DEEP LEARNING.

Predicted bug locations by deep learning				Real bug locations	
Library	Application	File Name	Line Number	File Name	Line Number
libming	listaction_d	decompile.c	103,370,381,407,440,455,476,477,555,569,583,714,726,762,1597,1690,1715,1932	decompile.c	440,455,370,381,714
	listswf_d			decompile.c	370,440,407,477,455,381,476,1932,714,762,726,103
	swftocxx			decompile.c	1597,440,370,381,455,477
	swftoperl			decompile.c	407,440,370,455,381,477,714,1690
	swftophp			decompile.c	477,370,381,455,440,407,714,762,103,726
	swftopython			decompile.c	714,103,762,569,583,555,726,583
gif	egif_lib.c	egif_lib.c	92,764,802,1144	egif_lib.c	771,790
	gif sponge	gif sponge.c	44,76,81	gif sponge.c	NA

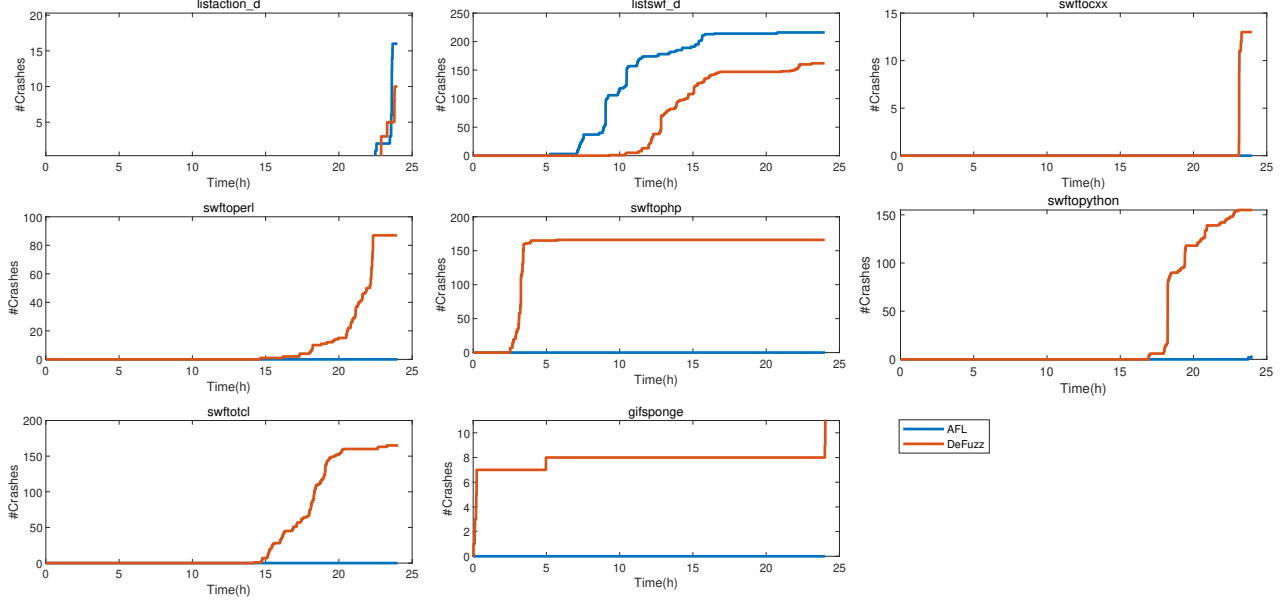


Fig. 5. The number of crashes over time. DeFuzz exposes crashes faster than AFL.

all the bug locations found by fuzzing on the seven applications from libming. On the application gif sponge, deep learning does not predict the accurate bug locations, however, the real bug locations are close to the predicted ones. For example, deep learning predicts that the bug locations include lines 764 and 802 in file *egif\_lib.c* while the real bug locations are the lines 771 and 790. When DeFuzz can successfully guide fuzzing to the predicted locations, it has a high chance that the neighbour code lines can also be executed. Therefore, DeFuzz can discover bugs that are close to the predicted lines.

On the other hand, AFL only detects bugs on two applications *listaction\_d* and *listswf\_d*. As shown in Table III, DeFuzz found 63 bugs while AFL found only 20 bugs. Meanwhile, DeFuzz exposes two types of bugs, *i.e.*, *out-of-memory* and *double free*. However, AFL only exposes bugs with the type *out-of-memory*.

### C. Limitations

In order to guide fuzzing to target locations, DeFuzz has to calculate the distance between the current execution path and the target locations. However, the calculation of distance takes much time. Although DeFuzz calculates the distance during

static analysis, which does not affect the execution speed of target programs, it may have problems when fuzzing on large programs. Moreover, DeFuzz uses control flow graph (CFG) to calculate distance. However, it is hard to statically construct an accurate CFG, which may miss some connections between two different basic blocks. The missed connections lead to an inaccurate distance.

Another limitation is that we usually face with the class imbalance problem with the data-driven cybersecurity. In this study, the ratio of vulnerable and not vulnerable samples we collected is about 1:16.5. In this case, the classifiers usually bias toward the majority class. For example, assume there are 95% of the data samples are from majority class (*e.g.*, not-vulnerable) while only 5% of the data samples are from the minority class (vulnerable). If the classifier classify all the data samples as not-vulnerable, the accuracy will be 95%, however, this is useless in the real-world security problem because we want to identify as many vulnerabilities as possible. Therefore, it is very necessary to address the class imbalance problem with data-driven cybersecurity problems. In our previous study, we developed DeepBalance [1] to address the class imbalance problem, however, there are still much room for improvement.



## V. CONCLUSION

In this study, we proposed a DeFuzz, which is a deep learning guided directed fuzzing for software bugs identification. The goal is to use deep learning to identify the potential vulnerable locations, then use directed fuzzing to fuzz the vulnerable locations to reduce the false positive rate. To achieve this goal, DeFuzz uses the attention-based BiLSTM to train a prediction model based on the ground truth data we collected. The prediction model is used to identify any potential vulnerabilities given an unknown test case. We then run AFLgo to fuzz the potential vulnerable locations. We have demonstrated that DeFuzz outperforms the baselines by conducting experiments on real-world data.

Although DeFuzz achieves better performance than the baseline, there are still several directions that one can further explore in the future. For example, deep learning can be utilised to guide fuzzing other than setting potential target locations. Deep learning can also be used to optimise the mutation scheme of fuzzing, such as how to mutate an input or the times an input to be mutated. Another interesting direction is to address the class imbalance problem when using the deep learning-based techniques.

## REFERENCES

- [1] Shigang Liu, Guanjun Lin, Qing-Long Han, Sheng Wen, Jun Zhang, and Yang Xiang. Deepbalance: Deep-learning and fuzzy oversampling for vulnerability detection. *IEEE Transactions on Fuzzy Systems*, pages 1329–1343, 2019.
- [2] Rory Coulter, Qing-Long Han, Lei Pan, Jun Zhang, and Yang Xiang. Code analysis for intelligent cyber systems: A data-driven approach. *Information Sciences*, 2020.
- [3] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):1–36, 2017.
- [4] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE*, 2020.
- [5] Xiaogang Zhu, Xiaotao Feng, Tengyun Jiao, Sheng Wen, Yang Xiang, Seyit Camtepe, and Jingling Xue. A feature-oriented corpus for understanding, evaluating and improving fuzz testing. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 658–663, 2019.
- [6] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.
- [7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.
- [8] Xinbo Ban, Shigang Liu, Chao Chen, and Caslon Chua. A performance evaluation of deep-learned features for software vulnerability detection. *Concurrency and Computation: Practice and Experience*, 31(19):e5103, 2019.
- [9] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.
- [10] Shigang Liu, Mahdi Dibaci, Yonghang Tai, Chao Chen, Jun Zhang, and Yang Xiang. Cyber vulnerability intelligence for internet of things binary. *IEEE Transactions on Industrial Informatics*, 16(3):2154–2163, 2019.
- [11] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Yang Xiang, Olivier De Vel, and Paul Montague. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Transactions on Industrial Informatics*, 14(7):3289–3297, 2018.
- [12] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344. ACM, 2017.
- [13] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018.
- [14] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.
- [15] Lwin Khin Shar and Hee Beng Kuan Tan. Predicting sql injection and cross site scripting vulnerabilities through mining input sanitization patterns. *Information and Software Technology*, 55(10):1767–1780, 2013.
- [16] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812. IEEE, 2015.
- [17] Henrique Alves, Baldoino Fonseca, and Nuno Antunes. Experimenting machine learning techniques to predict vulnerabilities. In *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, pages 151–156. IEEE, 2016.
- [18] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discover: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, 2016.
- [19] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 85–96, 2016.
- [20] Yao Li, Weiyang Xu, Yong Tang, Xianya Mi, and Baosheng Wang. Semhunt: Identifying vulnerability type with double validation in binary code. In *SEKE*, pages 491–494, 2017.
- [21] Stefano Calzavara, Mauro Conti, Riccardo Focardi, Alvise Rabitti, and Gabriele Tolomei. Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 528–543. IEEE, 2019.
- [22] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Olivier De Vel, Paul Montague, and Yang Xiang. Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Transactions on Dependable and Secure Computing*, 2019.
- [23] Shigang Liu, Jun Zhang, Yang Xiang, and Wanlei Zhou. Fuzzy-based information decomposition for incomplete and imbalanced data learning. *IEEE Transactions on Fuzzy Systems*, 25(6):1476–1490, 2017.
- [24] Shigang Liu, Guanjun Lin, Lizhen Qu, Jun Zhang, Olivier De Vel, Paul Montague, and Yang Xiang. Cd-vuld: Cross-domain vulnerability discovery based on deep domain adaptation. *IEEE Transactions on Dependable and Secure Computing*, 2020.
- [25] Hongfa Xue, Shaowen Sun, Guru Venkataramani, and Tian Lan. Machine learning-based analysis of program binaries: A comprehensive study. *IEEE Access*, 7:65889–65912, 2019.
- [26] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 874–889, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.
- [27] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. Greyone: Data flow sensitive fuzzing. In *USENIX 2020*, 2020.
- [28] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2018.
- [29] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: fuzzing deeply nested branches. In *CCS*, 2019.
- [30] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, and Yu Song. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, 2019. USENIX Association.
- [31] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspon-

dence. In *The Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2019.

- [32] X. Zhu, X. Feng, X. Meng, S. Wen, S. Camtepe, Y. Xiang, and K. Ren. CSI-Fuzz: Full-speed edge tracing using coverage sensitive instrumentation. *IEEE Transactions on Dependable and Secure Computing*, 2020.
- [33] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.
- [34] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.
- [35] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.
- [36] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.
- [37] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Long Lu, et al. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.
- [38] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association.
- [39] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *NDSS*, 2020.
- [40] Libming wiki. <http://www.libming.org/>, 2020.
- [41] aflutils. Utilities for automated crash sample processing/analysis, easy afl-fuzz job management and corpus optimization. <https://gitlab.com/rc0r/afl-utils/>, 2020.
- [42] Gdb: The gnu project debugger. <https://www.gnu.org/software/gdb/>, 2020.