An Iootfuzzer Method for Vulnerability Mining of Internet of Things Devices based on Binary Source Code and Feedback Fuzzing

Guangxin Guo, Chao Wang*, Jiahan Dong, Bowen Li, Xiaohu Wang
State Grid Beijing Electric Power Research Institute, Beijing, China

***Abstract*—With the technological progress of the Internet and 5G communication network, more and more Internet of Things devices are used in it. Limited by the cost, power consumption and other factors of Internet of Things devices, the systems carried by the Internet of Things devices often lack the security protection provided by larger equipment systems such as desktop computers. Because the current personal computers and servers mostly use the x86 architecture, and the previous research on security tools or hardware-based security analysis feature support is mostly based on the x86 architecture, the traditional security analysis techniques cannot be applied to the current large-scale ARM-based and MIPS-based Internet of Things devices. Based on this, this paper studies the firmware binary program of common Linux-based Internet of Things devices. A binary static instrumentation technology based on taint information analysis is proposed. The paper also analyzes how to use the binary static instrumentation technology combined with static analysis results to rewrite binary programs and obtain taint path information when binary programs are executed. Firmware binary fuzzing technology based on model constraints and path feedback is studied to cover more dangerous execution paths in the target program. Finally, iootfuzzer, a prototype vulnerability mining system for firmware binaries of Internet of Things devices, is used to test and analyze the two technologies. The results show that its fuzzing efficiency for Internet of Things devices is better than other fuzzing technologies such as boofuzz and Peach 3. It can fill in some gaps in the current security analysis tools for the Internet of Things devices and improve the efficiency of security analysis for Internet of Things devices, which contributes to the field through automated security vulnerability detection systems.**

***Keywords*—Internet of things; system vulnerabilities; source code; fuzz testing; instrumentation technology**

I. Introduction

At present, the interaction between IoT devices and the outside world is mostly carried out through the network. Usually, the software monitors the input data of external users, and the user's operation on the device is received and processed through several specific softwares [1]. Therefore, to analyze the vulnerability of this software, it is necessary to start from the code path through which external user data flows and find the problem code that external users, or attackers, can reach [2].

At the same time, in the past commonly used fuzzy testing based on path feedback, almost all of them adopt the way of the full instrumentation, such as path record instrumentation for all jump instructions at the end of the code block [3]. This instrumentation method will lead to a lot of programs internal processing codes unrelated to user input being instrumented, and then will make the code unrelated to external input data generate a new execution path so that the fuzz test tool will mistakenly think that the new execution path is caused by the fuzz test sample, and use the fuzzing test sample to further mutate and test in the follow-up. This will reduce the efficiency of fuzz testing.

At present, the work of source code vulnerability mining mostly depends on manually defined rules, but this method has some drawbacks: on the one hand, manually defined vulnerability mining rules often need to rely on the expertise and work experience of experts [4]. It is difficult to ensure full coverage of the possible causes of vulnerabilities, and it is easy to have the possibility of false positives and underreporting. On the other hand, it takes a lot of manpower to mine loopholes according to the rules. Due to manual judgment, the phenomenon of false underreporting will still occur [5]. With the development of technology, researchers began to use machine learning methods for vulnerability mining. This method does not need to define vulnerability rules, but still needs to define vulnerability features. Although it has reduced the manual workload, there are still drawbacks to feature coverage. In recent years, with the increasing popularity of deep learning, many scholars have begun to try to apply deep learning methods to vulnerability mining, and have made good progress [6]. In the process of data preprocessing, only whether the source code contains vulnerabilities is divided, and the types of vulnerabilities are not classified, so the existing work can only detect whether a piece of code contains vulnerabilities, and cannot accurately detect the types of vulnerabilities.

Scandariato et al. [7] tested the bag-of-words technique with a hybrid approach combining N-gram parsing and statistical feature selection to predict vulnerable software components. Yamaguchi [8] proposed a new graphical representation, called the code property graph, by traversing the graph to discover vulnerabilities. However, designing effective traversals to detect complex vulnerabilities can be very difficult. Thus, the authors propose an automatic method for traversing code attribute graphs to effectively locate taint-style vulnerabilities generated by uncleaned data streams, and use it to experiment with five popular open-source projects. The number of source codes without vulnerabilities in the data set is much larger than the number of source codes with vulnerabilities. It is difficult to learn the characteristics of a

*Corresponding Author.

small number of samples, while a large number of samples are prone to over-fitting during training.

To solve the problem that it is difficult to apply the fuzzing technology based on path feedback to the firmware binary program of ARM and MIPS Internet of Things devices and the efficiency of fuzzing test is low, this paper proposes a binary static instrumentation technology based on the taint information analysis. Firmware binary fuzzing testing technology based on model constraints and path feedback is studied in depth. The binary static instrumentation technology based on taint information analysis is studied, which can solve the problem that the current ARM and MIPS architectures lack taint information analysis tools. It analyzes and instruments the conditional branch jump of the target program in firmware affected by external input data to provide information feedback to the fuzz testing tool. The firmware binary fuzzing testing technology based on model constraints and path feedback is studied, which can improve the efficiency of fuzzing testing technology such as model constraints and path feedback combined with the above technology applied to the vulnerability mining of the target device, and focus the fuzzing test on the dangerous path to achieve the effect of improving the efficiency of fuzzing testing. The abstract syntax tree of function is used to represent the function, and multiple functions are annotated in open source datasets to capture the intrinsic representation of vulnerabilities, which proves that the model is effective for cross-project vulnerability detection at the functional level. Finally, the effectiveness of the above technology is proved by comparative experiments.

The main innovations of this paper are:

1) Adopt a feedback type fuzz test technology to carry out vulnerability mine on that firmware of the Internet of things equipment, and select the conditional branch jump points influenced by external user input data, namely taint information, as path feedback data of the fuzz test.

2) Design and implement a firmware vulnerability mining prototype system based on binary static instrumentation and feedback fuzzing for ARM and MIPS architectures.

3) Based on the results of taint information analysis, the target binary program is instrumented to improve the execution efficiency of the fuzz testing process.

Through the binary static technology based on the Internet of Things device vulnerability analysis, the fuzzing testing tool can obtain the feedback information of the relevant test samples in the process of fuzzing testing, and guide the generation of the fuzzing test samples. At the same time, the relevant algorithm is designed to select more valuable samples, and the model constraints are used to make the fuzzy test towards a higher coverage.

II. Related Work

A. Binary Static Instrumentation Technique

Static Binary Instrumentation (SBI) modifies the binary program file stored in the storage medium to generate an instrumented binary file and save it to the storage medium. When the program is executed, the instrumented binary file is run [9]. The idea of binary static instrumentation technology is very simple, which is to modify the binary file through the target file format to achieve the purpose of adding specific new code, but it needs to statically analyze the instrumented program in advance or describe it through the configuration file [10]. One of the core parts of binary static instrumentation technology is the selection of instrumentation positions. An example of an instrumentation error is shown in Fig. 1.

Static instrumentation technology modifies and hijacks the original code execution flow of the program so that the program jumps to the instrumentation code to execute when it runs to a specific location and completes the specific functions inserted by developers [11]. The selection of the insertion point will not only affect the efficiency of the program execution but also affect whether the program can be executed normally. In instrumentation tools that use binary static instrumentation techniques, directly editing the target binary file format and modifying and inserting code is the most common implementation.

B. Feedback Fuzzy Test Technique

Under the condition that the source code can be instrumented, AFL, honggfuzz, and other fuzzing tools all adopt the feedback fuzzing technology. These fuzz testing tools use the execution path information of fuzz test samples as feedback information to guide the sample generation tool to generate samples that can improve the coverage of fuzz test code (path) [12]. AFL uses customized GCC to insert the functional code of path recording and path feedback into the compiled binary program through source code instrumentation so that the program can record and feedback the execution information of fuzzy test samples during running [13]. In the absence of source code, binary instrumentation is generally used to obtain the key information in the running process of the target program, which requires researchers to develop and customize it for specific situations. Under the condition of no source code, AFL can still fuzz binary programs with the QEMU tool, but the efficiency of fuzz testing is low [14]. InsFuzz uses binary static instrumentation to interpolate non-source binary programs, inserting path records and feedback code into binary programs to achieve the same effect as source instrumentation [15]. Through the feedback fuzzing testing technology, the efficiency of fuzzing testing can be effectively improved. Inefficient fuzzing test samples can be discarded in time, and only the fuzzing test samples which may generate new execution paths can be mutated so that the whole fuzz testing work can be carried out in the direction of improving the code test coverage.

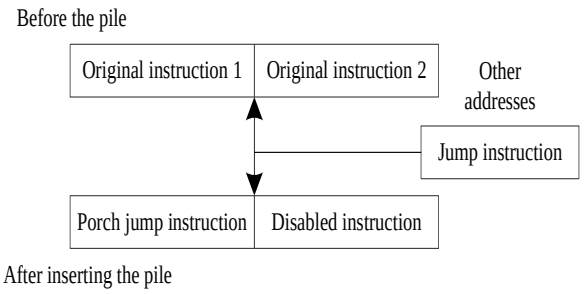

Fig. 1. Example of CISC Instrumentation Causing Program Runtime Errors.

III. Research on Binary Static Instrumentation Technology based on Stain Information Analysis

C. Analysis Flow and Algorithm of Taint Information

To improve the effective code coverage rate of the subsequent fuzz testing tool, the technology performs simulation execution on the target binary program, and therefore, more functional codes need to be analyzed as much as possible in the taint information analysis process. Because the target binary program in this paper is the network program in the firmware of the Internet of Things device, which uses socket, bind, accept, fgets, send, and other functions to build the data receiving and sending part of the network application program. These programs all have a clear function to receive external input data. The Internet of Things device firmware program uses the fgets function to receive external data packets transmitted through the network through the file descriptor of the socket.

Since program codes are executed in a simulation execution mode when taint information analysis is carried out [16], even if the analysis is not purely static, partial run-time information is missing. The coding fragments are shown in Table I.

TABLE I. Code Fragment

Code fragment:
<META HTTP-EQUIV="0,0x271t" CONTENT="no-cache"> If iggate,10000,(ffile * dword_654BC V0=400; V1=”Bad Request”; V2=”No request found” Return sub_D343（V0, V1, V2）; <META HTTP-EQUIV="Cache-Control" CONTENT="no-cache, must-revalidate">

As shown in the variable s in Table I, during simulation execution and taint analysis, it is impossible to accurately assign the external input variable or simulate the external input that can explore all execution paths [17]. Therefore, it is necessary to apply some methods to analyze the code that operates on the external input data as much as possible during simulation execution and taint analysis.

D. Research on Firmware Binary Fuzz Testing Technology for Internet of Things Devices based on Model Constraints and Path Feedback

- Fuzz testing coverage and sample selection algorithm

When the feedback information received by the sample variation module finds that an execution path is generated, it indicates that the current fuzzy test sample triggers a new execution path [18], which means that the change of some fields in the sample triggers the change of the flow direction of the program control stream, and by performing sample variation on the sample again and generating a new sample. There is a greater probability that the code coverage of the fuzz test can be improved, so it is placed in the queue of samples to be mutated.

In the whole process of fuzzing testing, this paper defines an index of dangerous branch jump coverage, which is used to record and judge the index data in the process of fuzzing testing:

$$C_{danger} = \frac{DB_{executed}}{DB_{all}} \times 100\% \tag{1}$$

In Formula 1, DB_{all} represents the number of branch jumps influenced by the external user input data in the target binary program, and $DB_{executed}$ represents the number of branch jumps influenced by the external user input data that have been executed in the current fuzz test.

A queue $Q = \{S_0, S_1, \cdots, S_n\}$ and a queue $M = \{m_0, m_1, \cdots, m_n\}$ of fuzzing test samples to be tested are defined for the fuzzing test samples transmitted to the fuzzing testing module. During the fuze testing process, the fuze test samples in the queue Q are transmitted to the fuzz testing module in a first-in first-out order [19]. If a new execution path is generated after the fuzz test samples are executed, then pass it to the use case generation function and add the newly generated fuzzing test sample to the fuzzing test sample queue M.

- Target feedback data processing

An array $DB = \{DB_0, DB_1, \cdots, DB_n\}$ is used to record the execution of the current fuzz test sample in the target binary program. This data is fed back after the target binary program executes the external input data processing function. At the same time, the array $AC = \{AC_0, AC_1, \cdots, AC_n\}$ is defined to record the total number of times that the branch jump of each taint condition is executed in the fuzz test so far.

As shown in Equation 2, a weight must be defined for each fuzzy test sample to represent the "value" of the fuzzy test sample:

$$\begin{cases} P_{ij}=1 \ \ (DB_{ij} \neq 0) \\ P_{ij}=0 \ \ (DB_{ij} = 0) \\ W_i = \sum_{j=0}^{n} (\frac{P_{ij}}{AC_j} \times 1000) \end{cases} \tag{2}$$

The weight of each branch is $1000 / AC_i$. When a taint condition branch jumps in the fuzz test and the total number of times it has been executed so far is AC_i, its reciprocal is multiplied by 1000 to get its weight. The factor of 1000 is used to prevent too many executions from causing the reciprocal to be too small, and the value becomes 0 after the decimal place is normalized. The factor can increase with the number of fuzz tests without affecting the weight ordering.

IV. Experimental Analysis

E. Function Comparison of Stain Analysis Tool

At present, there is no mainstream tool of the same type supporting ARM and MIPS that is open source or available for download for comparison, as shown in Table II.

TABLE II. Comparison Results of Stain Analysis Tools

Tool name	x86	arm	mips	Required documents
Pin	Y	N	N	Source Code
Taintgrind	Y	Y	N	Source Code
TaintEraser	Y	N	N	Source Code
TEMU	Y	N	N	Source Code
PyDaint	Reserved interface	Y	Y	Binary

Most mainstream tools rely on the source code to recompile the analysis target before running the analysis. PyDaint not only supports the ARM and MIPS architectures studied in this paper, but also can be extended to further support x86 architectures [20].

The test framework is shown in Fig. 2.

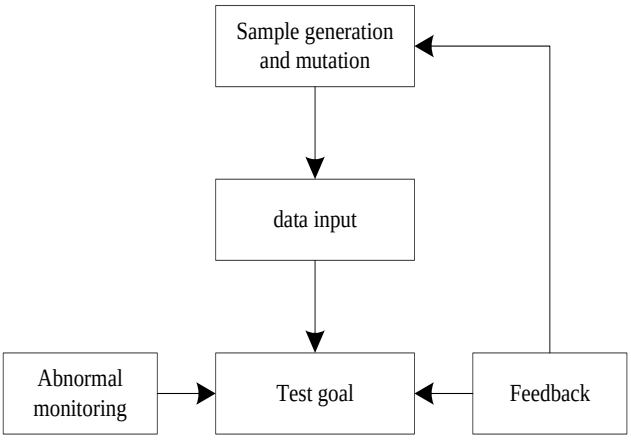

Fig. 2. Basic Framework of Paste Test.

F. Instrumentation Performance Test of Tainted Information Flow

Because this project uses QEMU user mode to run simulation on x86 _ 64 computers for ARM and MIPS architectures, it is difficult to carry out relevant timing statistics. Therefore, to test the effect of binary static instrumentation based on taint information analysis on the execution efficiency of the original binary program, the Web service programs of ASUS AC88U router based on ARM architecture and D-LinkDIR882 router based on MIPS architecture are instrumented respectively. It takes for normal user interaction (simulated access and packet reception through a Python program) before and after instrumentation in a test environment. The experimental results are shown in Table III and Table IV.

TABLE III. Comparison of Execution Efficiency before and after HTTPD Instrumentation in ARM Architecture

Test object / Sample number	Before insertion	After insertion
100	432.58ms	846.39ms
1000	4545.74ms	8682.94ms

The average time for httpd to send and receive data is about 4.3ms before instrumentation, and 8.7ms after instrumentation. In the experimental environment, the instrumentation loses about 1.02 times of performance.

TABLE IV. Comparison of Execution Efficiency of Lighttpd before and after Instrumentation of MIPS Architecture

Test object / Sample number	Before insertion	After insertion
100	32.59ms	96.34ms
1000	228.39ms	972.21ms

The average time for lighttpd to send and receive data is about 0.28ms before instrumentation, and 0.99ms after instrumentation. In the experimental environment, the instrumentation loses about 2.5 times of performance.

G. Fuzzy Test Function Comparison

Before analyzing the experimental results of fuzz testing efficiency, as shown in Table V, we first compare the functions of several common fuzz testing tools with the fuzz testing tool iboofuzzer implemented in this paper.

TABLE V. Comparison Results of Fuzz Test Tools

Tool names	Network Program Fuzziness Test	Path feedback	Path feedback without source code	Model constraints
AFL	N	Y	Partially supported	N
AFL-net	Y	Y	Partially supported	N
Peach 3	Y	N	N	Y
boofuzz	Y	N	N	Y
iboofuzzer	Y	Y	Y	Y

Iboofuzzer is a fuzzy testing subsystem developed for the network binary program in the Internet of Things device firmware in this paper, which supports model constraints and path feedback, and can obtain the feedback information of the target binary program by using instrumentation tools in common use scenarios without source code.

H. Fuzzy Test Efficiency Test

When using the original boofuzz for fuzz testing, the sample random mutation function is added to avoid the premature end of the fuzz test due to sample exhaustion. At the same time, the iboofuzzer framework is used to count the coverage information of boofuzz and Peach 3 tests, but it is not fed back to the sample generation module for analysis and comparison. In the testing process, the original boofuzz is tested based on model constraints and random data. For Peach

3, its listening mode is used, and the fuzzy test samples are actively obtained from Peach through the framework of iboofuzzer and then sent to the target binary program. The test parameters are shown in Table VI.

TABLE VI. FUZZY TEST TOOL EXPERIMENTAL VARIABLES

Tool names	Sample generation method	Information feedback
boofuzz	Random data	No
boofuzz	Model constraints	No
Peach 3	Model constraints	No
iboofuzzer	Model constraints	Path feedback

For each of the four test conditions in Table VI, a blur test was performed for about 8 hours.

- Comparison of coverage rate of insertion point in fuzzy test

The coverage rate of instrumentation points in the fuzzing test process represents the proportion of different branches of instrumented points executed in the whole fuzzing test process, as shown in Fig. 3.

In this comparative experiment, the coverage of boofuzz, which uses random data generation for fuzz testing, is the lowest in the whole 8-hour test. The second-lowest is boofuzz, which uses model constraints to generate samples. It is lower than Peach 3 in the first few hours, and then gradually approaches. Peach 3 has the second-highest coverage and iboofuzzer has the highest coverage. That is to say, the quality of samples generated by iboofuzzer is higher, and the test coverage of dangerous paths is larger. In the experimental environment, the coverage rate of the algorithm in this paper is the highest, and the target binary program after interpolation can still process a single fuzzy test sample in milliseconds.

- Comparison of the number of new execution paths for fuzz testing

The number of new execution paths generated during fuzz testing represents the ability of the fuzz testing tool to find new code test paths throughout the fuzz testing process, which is shown in Fig. 4.

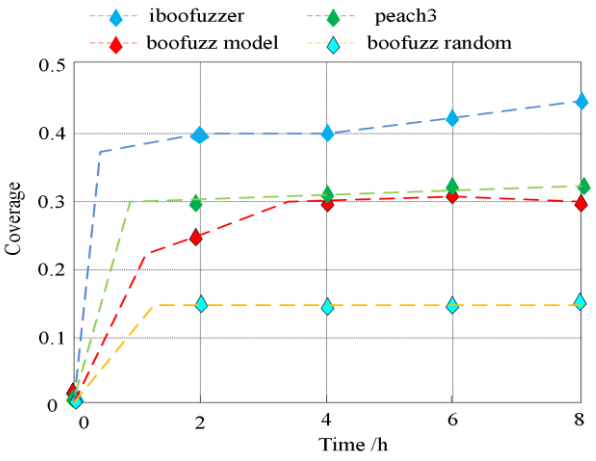

Fig. 3. Comparison of Fuzz Test Instrumentation Point Coverage.

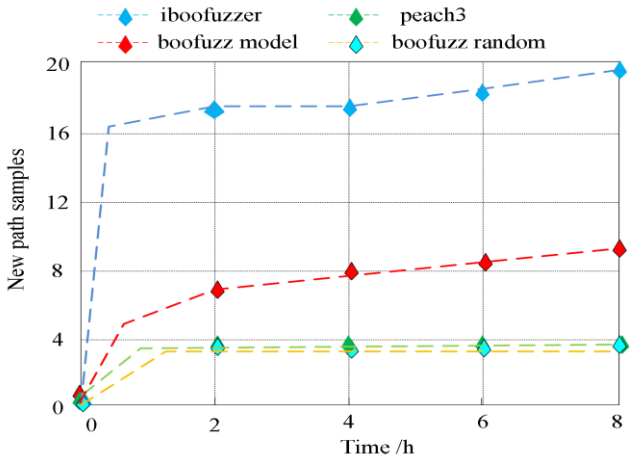

Fig. 4. Comparison Chart of the Number of New Execution Paths for Fuzz Testing.

Peach 3 and boofuzz with random data both generate fewer new path samples, while iboofuzzer generates the newest path samples, that is, the samples generated by iboofuzzer are more effective in discovering new paths. The distortion efficiency is improved by increasing the proportion of the effective distortion in the total distortion. Samples with effective distortion can reach the target basic block faster, while samples with invalid distortion will make the program fall into the situation of path explosion.

The object of vulnerability mining in this paper is the network binary program in the Internet of Things devices, which has certain format requirements for the input data, so the generation of samples based on model constraints can not only solve the problem of lack of original data samples in fuzzy testing, but also improve the code penetration of samples, and avoid the samples being abandoned in the format check function of the target program. At the same time, the fuzzy test focuses on the dangerous path affected by the external input data, and mutates new fuzzy test samples to improve the coverage of the dangerous path. To improve the efficiency of fuzzy testing, the path feedback information and the calculated sample weight are used to guide the mutation generation of fuzzy testing samples.

V. CONCLUSION

In this paper, we study the static instrumentation of the firmware binary program of the Internet of Things device based on ARM and MIPS architecture, and innovatively combine the taint information analysis with the binary instrumentation and apply it to the firmware program of the Internet of Things device to improve the efficiency of fuzzing the dangerous path in the tested program of the firmware program of Internet of Things devices. The main work includes:

1) Analyze the taint information with the firmware binary program of the Internet of Things device based on ARM and MIPS architecture.

2) The feedback fuzzing technology is applied to the firmware binaries of the IoT devices based on ARM and MIPS architectures, and the target binaries are moved from the IoT

devices to desktop computers for fuzzing by using QEMU open source tools and binary static instrumentation technology.

3) Select some of the common security tools to compare with the subsystems of the prototype system iootfuzzer in different types, and use the tools that can fuzze the research goal of this paper to carry out the comparison experiment of fuzzing efficiency.

The fuzz testing subsystem in this paper is implemented based on boofbzz, and it needs to write samples to generate template files for different test objectives, which is a heavy workload. In the future, we can consider implementing the technology of automatically generating the corresponding template through the captured network communication packets to reduce the cost of preparation work in the early stage of fuzz testing. Complex vulnerabilities usually have a long ROC chain. These vulnerabilities may not only be related to one file, but also to multiple files, which will greatly affect the effect of vulnerability mining. Dynamic analysis is to detect vulnerabilities in the process of program running, which makes dynamic analysis more complex. In the future, the above two directions will be studied in depth.

REFERENCES

[1] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wfee Ng, Jolin Grundy, and Aditya Ghose. Automatic feature learning for vulnerability prediction. arXiv preprint arXiv: 1708.02368, 2017.

[2] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Out Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv: 1801.01681, 2018.

[3] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H. Deng. Vurle: Automatic vulnerability detection and repair by learning from examples. In European Symposium on Research in Computer Security, pages 229-246, 2017.

[4] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In 2018 17th IEEE49International Conference on Machine Learning and Applications (ICMLA), pages 757-762, 2018.

[5] Miltiadis Allamanis. Earl T. Barr, Premkumai Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR), 51(4):81, 2018.

[6] David Evans. Splint-secure programming lint. Technical report, Teclmical report, 2002. David Wheeler. Flawfinder home page, 2006, 2019-09-27.

[7] 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):56, 2017.

[8] Reudismam Rolim, Gustavo Soares, Ro hit Gheyi, Titus Barik, and Loris D'Antoni. Learning quick fixes from code repositories. arXiv preprint arXiv: 1803.03806, 2018.

[9] Zheng Y, Davanian A, Yin H, et al. FIRM-AFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation[C]//28th {USENDC} Security Symposium ({USENIX} Security 19). 2019: 1099-1114.

[10] Fowler D S, Bryans J, Shaikh S A, et al. Fuzz testing for automotive cyber security[C]//2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W). IEEE, 2018: 239-246.

[11] David Liu, Andrew Petersen. Static Analyses in Python Programming Courses. 2019:666-671.

[12] Zhao Na. Research on the role of new multilateral development financial institutions in the construction of the “Belt and Road”. Journal of Economic Research. 2020(10).

[13] Zhang Fengyang. Difficulties and Suggestions of Financial Consumer Protection in Rural Financial Institutions. Business News. 2020(12).

[14] Zhu Jie, You Xiong, Xia Qing. Space-time data organization model of battlefield environment objects based on mission process. Journal of Wuhan University (Information Science Edition). 2018(11).

[15] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, et al. A Survey of Symbolic Execution Techniques. ACM Computing Surveys, 2018, 51(3):1-39.

[16] Caroline Lemieux, Koushik Sen. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. 2018:475-485.

[17] M. A. Klimushenkova, P. M. Dovgalyuk. Improving the performance of reverse debugging. Programming and Computer Software, 2017, 43(1):60-66.

[18] Seyed Mohammad Ghaffarian, Hamid Reza Shahriari. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques. ACM Computing Surveys (CSUR), 2017, 50(4).

[19] Philipp Dominik Schubert, Ben Hermann, Eric Bodden. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. 2019, 11428:393-410.

[20] Shuai Wang, Dinghao Wu. In-memory fuzzing for binary code similarity analysis. 2017:319-330.