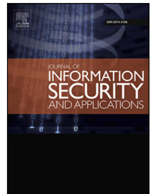
Journal of Information Security and Applications 54 (2020) 102497

Contents lists available at ScienceDirect

Journal of Information Security and Applications

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



AFLPro: Direction sensitive fuzzing

Tiantian Ji[a], Zhongru Wang[a,b,*], Zhihong Tian[c], Binxing Fang[a,c], Qiang Ruan[d], Haichen Wang[e], Wei Shi[f]

[a] *Key Laboratory of Trustworthy Distributed Computing and Service (BUPT), Ministry of Education, Beijing University of Posts and Telecommunications, Beijing, China*
[b] *Chinese Academy of Cyberspace Studies, Beijing, China*
[c] *Cyber space Institute of Advanced Technology, Guangzhou University, Guangzhou, China*
[d] *Beijing DigApis Technology Co., Ltd, Beijing, China*
[e] *Beijing University of Posts and Telecommunications, Beijing, China*
[f] *School of Information Technology, Carleton University, Ottawa, Canada*

ARTICLE INFO

Keywords:
Automated binary fuzzing
Direction sensitive fuzzing
Basic block aggregation
Seed selection
Seed energy scheduling
Static analysis

ABSTRACT

Fuzzing is a simple and popular technique that has been widely used to detect vulnerabilities in software. However, due to its blind mutation, fuzzing brings many limitations. First, it is difficult for fuzzing to pass the sanity checks, which makes fuzzing unable to target vulnerability or crash locations effectively. Secondly, blind mutation limits the diversity of seed generation and makes it difficult for the fuzzing process to achieve convergence.

In this paper, we propose a direction sensitive fuzzing solution AFLPro. On the one hand, it focuses on seed selection, using a new fuzzing scheme based on Basic Block Aggregation (BBA), which reduces the possibility of seed selection in the wrong direction. By applying a multi-dimensional oriented seed selection strategy, it achieves fine-grained seed selection. On the other hand, based on biological evolution, AFLPro optimizes genetic variation to ensure the diversity of seed varieties and the convergence of fuzzing tests. Besides, AFLPro also incorporates lightweight static analysis to obtain information about the target program (this paper only studies closed source programs), providing complete semantic guidance for fuzzing through resource integration.

We implemented a prototype of AFLPro based on the popular fuzzer AFL. We evaluated it on three datasets: DARPA Grand Challenges (CGC), LAVA-M dataset, and a set of real-world applications. The results show that in 92% of all three datasets, AFLPro exhibits better vulnerability detection capabilities than all of the state-of-the-art fuzzers mentioned in this paper.

© 2020 Elsevier Ltd. All rights reserved.

1. Introduction

Fuzzing is an effective technology in software security testing. Recently, it is adopted to discover software vulnerabilities [1]. According to whether the prior knowledge of software code analysis is needed, fuzzing is categorized into black-box fuzzing, white-box fuzzing, and grey-box fuzzing. Comparing to the other two categories, grey-box fuzzing is more effective and its resource cost is smaller due to the utilization of lightweight code analysis. Thus, grey-box fuzzing is the most popular and effective vulnerability detection technology for practical applications to date. Specifically, since the creation of AFL [2], a representative of grey-box fuzzers, hundreds of high-risk vulnerabilities have been detected. However, grey-box fuzzing is still limited by its blind mutation. For example, given a random initial seed, it may take about 2^{5*8} mutations to pass the sanity check in the *if b == "heil0"* statement. A large number of unsuccessful sanity checks result in a shallow detection success rate. Furthermore, the input used by the fuzzer is constructed randomly, and the detection of the abnormal position is blind so that the existing method cannot achieve effective direcion sensitive fuzzing. Invalid guidance usually causes the fuzzer to discard mutation seeds that have significant contributions, which makes it difficult for the fuzzer to penetrate the program code, resulting in limited seed diversity and the fuzzer challenging to converge to the abnormal state point. In summary, passing the sanity checks in the target software efficiently and promptly remains a significant challenge.

* Corresponding author.
E-mail addresses: jitiantian0728@gmail.com (T. Ji), wangzhongru@bupt.edu.cn (Z. Wang), tianzhihong@gzhu.edu.cn (Z. Tian).

https://doi.org/10.1016/j.jisa.2020.102497
2214-2126/© 2020 Elsevier Ltd. All rights reserved.

Researchers have proposed many solutions to pass or bypass sanity checks in target software. These solutions mainly fall into three categories:

1. The solution that takes the symbolic execution as an assisting technology to the fuzzing: Driller [3] and T-Fuzz [4]. When fuzzing gets "stuck", the solution leverages symbolic execution to solve the path constraint problem to help fuzzers pass sanity checks in the target software.
2. The solution that focuses on using taint analysis technology to assist the fuzzing. Representative fuzzers in this category include Taintscope [5] and VUzzer [6]. These fuzzers pass sanity checks in two steps: first ranking input test cases; second, using a custom priority rule to guide the mutation process. However, symbolic execution and taint analysis are considered heavyweight program analysis techniques, which impose restrictions on the fuzzing process, such as higher resource consumption and significantly reduced scalability.
3. The solution that focuses on static analysis based fuzzing: VUzzer [6] and InsFuzz [7]. The static analysis based methods extract useful semantic information from the target program to guide fuzzers to pass security checks so that it can improve the fuzzing code coverage. Although this solution is not as expensive in terms of resource consumption as the heavyweight analysis solution described above, it is still limited in its vulnerability detection capabilities. The main reason is that this type of solution does not change the status of the localized orientation (Note: This article uses the term "orientation" to refer to the sensitivity to the correct direction of fuzzing) of fuzzing, which often leads to deviation from the correct fuzzing direction in the process of vulnerability detection. To dominate the right fuzzing path as much as possible to find the vulnerabilities in the target program accurately, we need a global-oriented strategy to guide the fuzzing during the vulnerability detection process. However, existing solutions using blind mutations in fuzzing do not yield the right global fuzzing directions. Therefore, how to guide the fuzzing going to an ideal global path constitutes another challenge.

In total, we summarize two main problems faced by fuzzing and its existing improvement schemes: (1) The issue of inability to effectively oriented towards vulnerability or crash positions due to sanity checks; (2) The limitation on the diversity of mutant seeds when it is not effectively oriented, and the problem that fuzzing test is difficult to converge. To address these two problems, we propose a direction sensitive fuzzing solution AFLPro. We focus our attention on seed selection and implement improvements around other genetic mutation operations related to seed selection. Specifically, this paper makes the following contributions:

- **We have developed a new fuzzing solution based on Basic Block Aggregation (BBA).** Through the unique analysis of the parent node, we conclude that the error-basic block usually corresponds to multiple parent nodes. Our solution reduces the importance of the error-basic block by Basic Block Aggregation (BBA for simplicity) and increases the importance of the parent node corresponding to the basic block. By adjusting the importance of the basic block on the current execution path, the possibility of selecting a seed on the wrong execution path can be reduced.
- **We propose a multi-dimensional oriented seed selection strategy.** From different dimensions, code coverage, local basic block weights, and global path weights are the three core factors that influence seed selection. Besides, seed length and seed execution time also affect the quality of seed selection to some extent. Based on the analysis of the influencing factors mentioned above, this strategy gives three principles for seed selection, which are used to achieve more fine-grained seed selection.
- **Optimizing genetic mutation: An energy scheduling strategy called Generations-Based Mutation (GBMutation) is designed.** According to the theory of biological evolution, generally, as the generation number of genetic mutation increases, the overall quality of the seed population is continuously improved. Therefore, using the depth of genetic mutation as one of the core factors of energy scheduling, this paper designs a GBMutaion-based seed energy scheduling strategy. Besides, this strategy introduces a fallback mechanism, which can speed up the convergence speed of the fuzzing test while ensuring seed diversity.
- **Information integration: Use very lightweight static analysis to obtain program information.** The static analysis based method can extract useful semantic information from the target program, mainly including two types of information: data flow information and control flow information. Through the resource integration of dynamic and static analysis information, the completeness of static analysis and the accuracy of dynamic analysis can be achieved at the same time, which provides a more robust orientation for fuzzing.
- **We implement a prototype AFLPro based on AFL** and evaluate it on three datasets: CGC[8] dataset, the LAVA-M[9] dataset, and a set of real-world applications. The results show that in 92% of all test programs, AFLPro performs better than state-of-the-art fuzzers in terms of vulnerability detection capability. It is worth noting that the target test objects in this paper are all closed-source binary programs.

2. Background and motivation

As a grey-box fuzzing tool with advantages of being fast, efficient and stable, AFL [8] has been widely recognized by both academia and industry. In this section, we illustrate the research motivation of this paper. We try to maintain the inherent advantages of AFL and implement a direction sensitive fuzzing solution based on AFL.

2.1. Motivation for oriented seed selection

To overcome the limitation of fuzzing blindness, seed selection and seed mutation all should be sensitive to fuzzing direction. The major problem for AFL to pass/bypass sanity checks is its blind mutation. AFL uses a fast algorithm to select a smaller subset of test cases that cover every branch of the test case tree. A branch that a seed executes is referred to as a *tuple*. As shown in Equ. 1, AFL classifies a seed as *favorite* if it is the fastest and smallest input for any of the tuples it exercises. However, in most cases, only the input with the "favorite" label will be selected as a next seed input. Consequently, according to the calculation of Equ. 1, unless the execution path of the current seed happened to be the correct test direction during fuzzing, all valid code that should have been examined becomes unreachable "dead code", which makes it difficult for AFL to locate the errors or vulnerabilities in the target program.

$$f_{afl} = t_i * l_i \tag{1}$$

As shown in Fig. 1, assuming the initial test case of AFL is the string "*Fuzz*", we conclude that there are two cases of error checking during fuzzing.

- There is a string matching check in module (a). We assume that AFL can mutate the initial seed into "*Fullo*" during fuzzing. Then both "*Fuzz*" and "*Fullo*" enter the error code block ***error-0***, and the execution paths of both two seeds are the same, so the seed execution time is considered to be the same. The

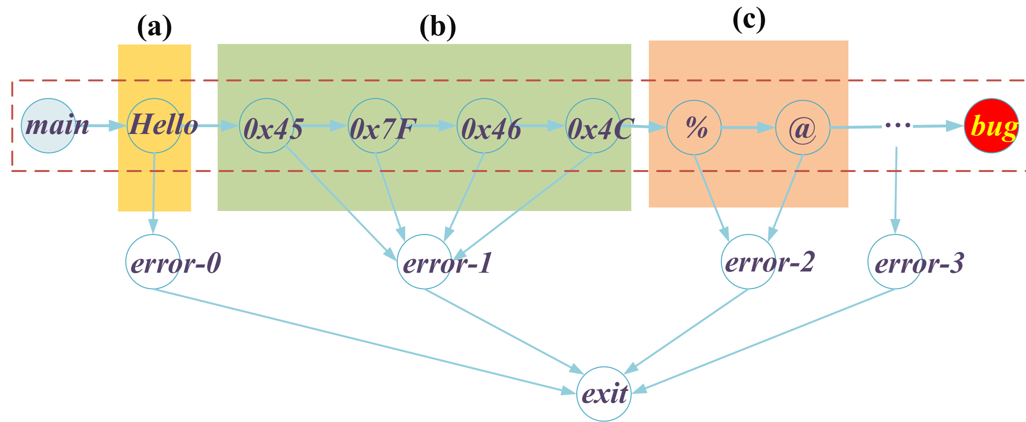


Fig. 1. An example of program control-flow.

length of "*Fuzz*" is shorter than the length of "*Fullo*", so when the Equ. 1 is used to select the optimal seed for the first tuple (block ***main*** to block ***Hello***, which we use (***main, Hello***) to represent) of the program, the AFL selects "*Fuzz*". But actually, whether from the perspective of program semantics or information entropy or Hamming distance, "*Fullo*" is a better seed than "*Fuzz*", and it is more likely for a mutation to produce the seed of "*Hello*".

- Suppose AFL is very fortunate to have two seeds that are "*Hello\x45\x7F\x46\x4C*" and "*Hello\x45\x7F\x46\x4C%*". Both of the two seeds trigger a new execution branch. When AFL selects a seed for the tuple (***0x46, 0x4C***), according to the product of execution time and seed length, the seed "*Hello\x45\x7F\x46\x4C*" with a smaller product value is selected, which leads AFL to deviate from the correct fuzzing direction.

After analyzing the above error checks, we draw several conclusions to support the seed selection: (1) The semantic information of AFL mutation results is important. (2) For the seeds that are executed on the same tuple, their next destinations are important.

Static Analysis. To guide the fuzzing direction, VUzzer adopts a path prioritization strategy, which calculates the basic block weight through static analysis and determines the importance of the execution path by the weight. Therefore, VUzzer alleviates the second situation of error checking mentioned above. This motivated us to integrate the static analysis technique into AFL. On the one hand, we can extract the data-flow features by static analysis to provide semantic information for the fuzzing. On the other hand, we can extract the control-flow features by static analysis to discriminate the importance of the path and provide better guiding support for the seed selection.

2.2. Motivation for oriented seed mutation

AFL uses a biological evolution-based cyclic feedback mechanism for fuzzing, and positive feedback is beneficial to the further testing of AFL. The quality of the feedback depends on seed mutation. During the seed mutation stage, the "good" mutation results are recorded and used for the mutation of the next generation. There are substantial differences between these recorded mutation results. For example, the importance of the paths they execute and the exploitable value of themselves are important factors influencing the mutation of the next generation. Therefore, for these "good" mutations, AFL gives different importance through energy scheduling. The higher the energy of a seed, the more critical the path it executes, and the more valuable its mutation results are. So the more energy a seed has, the more mutations it performs.

AFL divides the seeds into several different intervals based on their attribute values. And AFL assigns the same constant energy value to those seeds in the same interval based on the *Power Law*. The energy allocation strategy of AFL is based on the attribute characteristics of the seed, i.e., the exploitable value of the seed itself, without considering the importance of the path performed by the seed. Based on this, AFLFast [8] introduces the importance of the seed execution path, and it is considered that the seed on the low-frequency path is more important. However, to pursue test speed, AFLFast discards mutant seeds excessively, which makes it very easy to fall into a "stuck" state. This paper is inspired by AFLFast, and based on its work, we propose an improved energy allocation strategy that can better utilize the importance of path and the value of the seed itself.

3. Overview of AFLPro

To address the challenges mentioned earlier in this paper, we propose AFLPro, a direction sensitive fuzzer. Fig. 2 shows an overview of AFLPro, which consists of two major components: automated static analysis module and automated fuzzing module.

3.1. Automated static analysis module

In this module, the IDAPython [9] tool is used for automated static analysis. Additionally, the information can also be collected through the angr [10] tool, which integrates the static analysis technique. The input of this module is only the binary program without source code. At first, the Data-Flow Graph (DFG) and the Control-Flow Graph (CFG) are generated automatically in this module. Then we collect the Data-Flow Information (DFI) through DFG, including the byte information and the string information. At the same time, we also combine the basic block information obtained through CFG with the weight calculation model(as shown in Equ. 2) to get the basic block weight information, which we call Control-Flow Information (CFI) for clarity. The output of this module is DFI and CFI. They are stored in two different files. The format of one DFI and one CFI are expressed as follows, note that we use *BB* to represent a "Basic Block".

DFI string_*i*="XXXX"
CFI BB address, BB weight, branch BB address

Comparing with the performance consumption and time consumption of the entire vulnerability detection process, the performance consumption of the entire static analysis process is negligible, and the total time required for static analysis is calculated in seconds, which is also negligible. Thus, the cost of static analysis

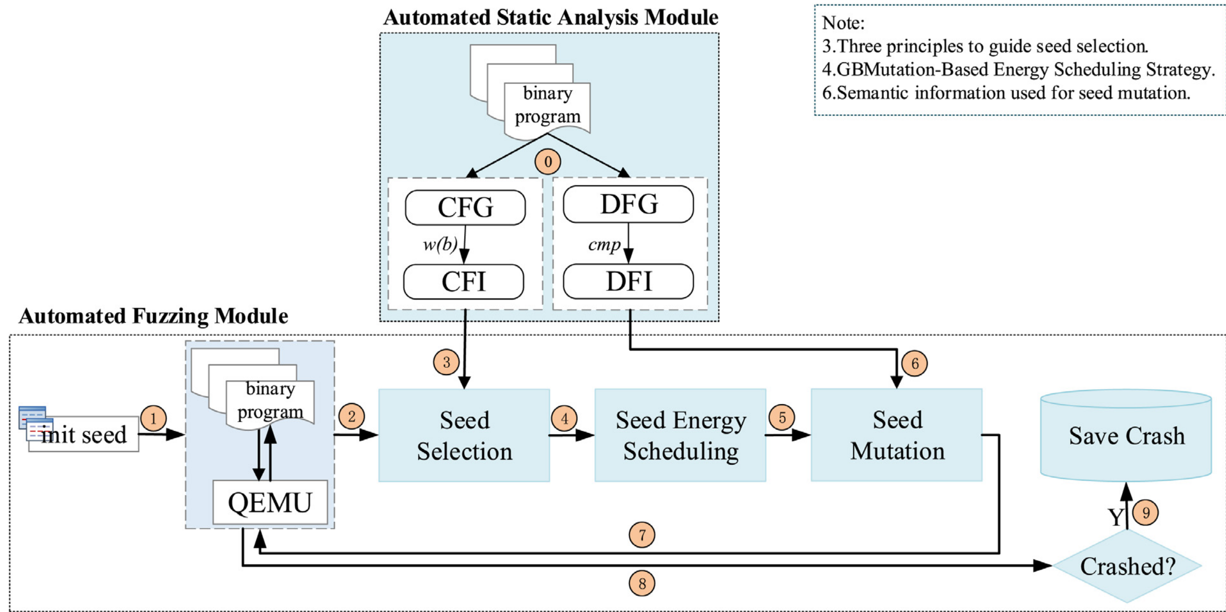


Fig. 2. A high level overview of AFLPro.

can be ignored in the whole fuzzing process, which is also an important reason why we choose the static analysis technique to aid the fuzzing.

3.2. Automated fuzzing module

The input and output of the automated fuzzing module are the same as AFL, that is, the input is the init seed and the output is the candidate input that may cause the program to crash. Apart from this, the input of this module also includes the DFI and the CFI provided by the automated static analysis module.

During fuzzing, this module gets seeds from the priority queue. In order to use the higher quality seed in the next fuzzing, it is necessary to adjust the priority order of the seeds in the queue. This module first makes a seed selection for each tuple with the guidance of the proposed BBA solution and the multi-dimensional oriented seed selection strategy. Specifically, this is achieved through the seed fitness calculation model, as shown in Equ. 3, which is built based on the CFI, and can provide better directional guidance for seed selection by considering local and global weight information.

After the seed selection, this module applies our seed energy scheduling strategy to guide the direction of the seed mutation with the expectation of providing high-quality feedback to the fuzzer. Our seed energy scheduling strategy is achieved through an energy function as well as a mutation strategy proposed in this paper, as shown in Equ. 4 and Fig. 3 respectively, which is a comprehensive consideration of the importance of the fuzzing path and the value of the seed itself.

The seed energy determines the number of seed mutation. After the seed energy scheduling, this module starts the seed mutation, in which we do not change the mutation strategies but provide useful semantic information for the seed mutation. The semantic information is the DFI provided by the automated static analysis module. We try to solve the first type of error checking problem encountered during fuzzing mentioned earlier by utilizing the seed mutation strategy based on program semantic information.

In summary, with the combination of the automated static analysis module and the fuzzing module, we propose a fuzzing improvement solution with BBA as the core design. This solution enhances the direcion sensitivity of fuzzing from three aspects: seed selection, seed energy scheduling and seed mutation, which are embodied in the multi-dimensional oriented seed selection strategy and the triple orientation of weight information, seed energy and semantic information.

4. Implement direction sensitive fuzzing

In this section, we will elaborate on the key technical details of AFLPro.

4.1. Basic block aggregation

In AFLPro, the basic unit of program analysis is the basic block or path branch (we call “tuple” for simplicity). We use the lightweight static analysis method to collect the basic block (sometimes we also use “node” to represent) information (i.e., CFI) of each function, including the inheritance relationship between basic blocks, the probability of generating each tuple, and the weight of basic blocks. By collecting these basic block information, we can help to determine the path direction, distinguish important paths, and assist dynamic seed selection in the process of dynamic fuzzing. This paper refers to the fixed point iterative algorithm mentioned in VUzzer to calculate the tuple weight or to calculate the fitness function in units of the tuple.

We improve the weight calculation model based on VUzzer, as shown in Fig. 1. In the general process of vulnerability detection, the path to the specific abnormal location of the program is usually unique, and the node in the fuzzing path usually has only one parent node. Especially for the high-level code language, taking a *if* statement (e.g.,

```
if(buf[1]==0x45 && buf[0]==0x7F &&
buf[3]==0x46 && buf[2]==0x4c)
```

) containing multiple judgments as an example, the *if* statement is displayed in the assembly code language as multiple paths consisting of multiple consecutive basic blocks or multiple tuples such as the module (b) in the Fig. 1. But in fact, the form of understanding of high-level language is more in line with the human way of thinking. If the judgment conditions of several consecutive basic blocks constituting the *if* statement are not all satisfied, then in the high-level language, the program is still in a state that has not passed the *if* condition. When the conditional checks of one

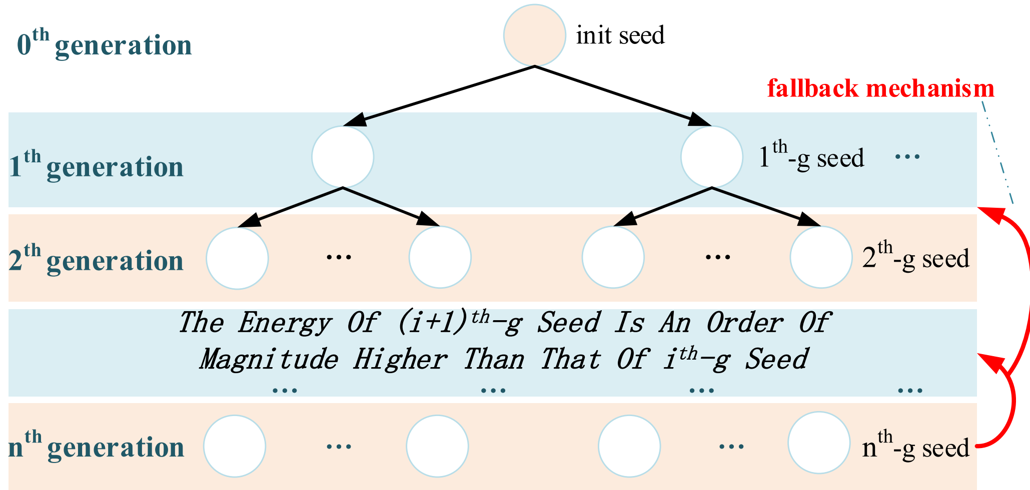


Fig. 3. The model of the GBMutation strategy.

or more consecutive basic blocks that make up the *if* statement fail, they all arrive at the same error-basic block, which usually has more than one parent node.

So based on the uniqueness analysis of the parent node, this paper proposes the idea of Basic Block Aggregation (BBA for simplicity). That is, by determining the parent node of the current basic block, for basic blocks with multiple parent nodes and basic blocks with fewer or a single parent node, we do different *log* processing for these two types of basic blocks and the specific processing operations follow the Equ. 2.

We define $\eta = \frac{1}{\sum_{c \in pred(b)} prob(b) * prob(e_{cb})}$ and $\delta = \frac{len(pred(b))}{len(pred(brob))}$ for clarity,

$$w(b) = \begin{cases} \eta * \log_2(len(pred(b)) + 1), & \delta < 1 \\ \frac{1}{\eta * \log_2(\delta + 1)}, & \delta > 1 \end{cases} \tag{2}$$

In the Equ. 2, *b* represents the current basic block; *brob* represents the sibling basic block of *b; pred(b)* represents the set of the parent basic blocks of basic block *b; c* represents a parent basic block of the current basic block *b; len()* is a function for length/number solving, *len(pred(b))* represents the number of *b*'s parent basic blocks; e_{cb} represents the tuple of *(c, b); prob(b)* means the probability of generating the basic block *b*, $prob(e_{cb})$ means the probability of generating the tuple *(c, b)*; finally, *w(b)* is the calculated weight of the current basic block *b*.

As shown in Fig. 1, the two modules **(b)** and **(c)** are the results of the BBA processing. By the BBA operations, the sibling basic blocks generated by the same parent node are compared. We assign less weight to the sibling basic blocks with more parent nodes by *log* processing and increase the weight of the sibling basic blocks with fewer parent node by a different *log* processing.

Taking the seed selection of the tuple (***0x4C, %***) in Fig. 1 as an example, suppose we have two kinds of seeds, the first type is going to the basic block ***error-2***, and the second type is going to the basic block ***@*** that satisfies the ***%*** judgment condition. Both the two seeds trigger a new tuple and increase the code coverage, which makes VUzzer difficult to distinguish the importance of the two seeds in both cases. But actually, the second type of seed is the seed in the correct test direction we want to choose. Based on this situation, we take use of the BBA idea, then the calculated weight of the *nextB* basic block ***%*** is significantly higher than the weight of the *nextB* basic block ***error-2***. Finally, we select the seed whose execution direction is the basic block ***%*** for the tuple (***0x4C, %***), which avoids the wrong test direction as much as possible. This paper focuses on seed selection. Our proposed BBA solution is the core design of seed selection. It can adjust the importance of the basic blocks on the current execution path to reduce the possibility of seed selection in the wrong direction. In the following, we will apply the BBA solution to the multi-dimensional oriented seed selection strategy.

4.2. Multi-Dimensional oriented seed selection strategy

Like most popular fuzzers, AFLPro is a code coverage-oriented fuzzer, that is, whether the state changes during program execution is based on whether the code coverage increases. However, based on our analysis above, during the fuzzing process, the increase in code coverage does not mean that it will be closer to the abnormal location of the program. We expect that each selected seed will not only increase code coverage but also be closer to the correct test direction, enabling deep penetration of the target program to increase the likelihood of trigger crashes. For this purpose, this paper proposes a more fine-grained seed selection scheme, namely Multi-Dimensional Oriented Seed Selection Strategy. The strategy takes the code coverage, local basic block weight and global path weight as three core dimensions, and takes the seed length and seed execution time into consideration. By considering different factors in multiple dimensions, we modeled the multi-dimensional oriented seed selection strategy (we call this model the seed fitness function) and designed three principles to be followed in the model establishment process. The details are as follows:

Principle 1. The most important thing is the next intention of all the seeds on this tuple, that is, to measure the importance of the target basic block of the next tuple to be executed by these seeds. For clarity, we use *nextB* to represent the target basic block of the next tuple. Therefore, the weight of *nextB* should be an important factor in the calculation of seed fitness on the tuple.

Principle 2. For the seeds that determine the same intention, the closer the seeds are to the abnormal position on the execution path, the higher the weight we assign to the seeds. Therefore, the weight of the seeds should be the second important factor in the calculation of fitness.

Principle 3. Finally, if the weights of some seeds are still the same, then the seed with the shorter length and less execution time is selected as the best seed on the tuple, and the correct test

direction in the seed selection process can be guaranteed as much as possible.

In the stage of seed selection on the tuple, AFL uses QEMU to perform instrumentation and simulation execution on binary programs. In this paper, only one instrumentation instruction is added based on AFL, which is used to obtain the actual memory address when the basic block is dynamically executed and can guarantee lower performance overhead and maintain the advantage of fast and stable. It is worth mentioning that when detecting a program that can cause QEMU to crash, the current vulnerability detection process will not be terminated due to the setting of the process management mechanism in AFL that makes QEMU and our target binary run in the same subprocess. In short, we should keep in mind that such programs that under test but can cause QEMU or AFL to crash do not affect the normal execution of fuzzing.

In this phase, this paper maps the basic block information collected in dynamic and static components. Then, based on Principle 1 and the idea of BBA, we leverage the proposed weight calculation model as shown in Equ. 2 to calculate the weight of the *nextB*.

After we select a class of seeds with the same local test direction according to Principle 1, we still have to make the best choice from such seeds. Therefore, this paper follows Principle 2 to add the weights of all the basic blocks on the path of each seed execution, which we use w_q to represent. And preferentially we select the seed q with a higher w_q. Compared with Principle 1, Principle 2 is a global optimization strategy which not only takes into account the importance of the entire path, but also a measure of the effective value of the seeds.

Principle 3 is a consideration of time and resource overhead in the fuzzing process. After ensuring the value of the seed and the direction of the fuzzing, we also need to consider the performance overhead in the fuzzing. So after the seed sifting by Principle 1 and Principle 2, we finally do the third selection, choosing the seed with the least performance overhead from the candidate seeds as the best seed.

Based on the above three principles, this paper proposes a tuple-oriented fitness function calculation model (as shown in Equ. 3). By calculating the fitness values of all the seeds passing through the same tuple, the seed with the highest fitness is selected as the best seed of the tuple.

$$f_{new} = \frac{[w(nextB) + \varepsilon] \cdot w_q}{log_2(t_q \cdot l_q)} \tag{3}$$

Note: ε is small enough to ensure that the molecule is not zero; *w(nextB)* represent the calculate result of the Equ. 2; t_q and l_q represent the execution time and the length of the seed q, respectively.

4.3. GBMutation-Based seed energy scheduling

According to the analysis of the impact factors of seed energy scheduling as mentioned before, AFL allocates seed energy according to *Power Law*, base on which, AFLPro develops the seed energy scheduling strategy. According to the theory of biological evolution, with the increasing generation number of cross-mutation, the biological population becomes more and more excellent. During fuzzing, the generation number of the seed mutation appears as the depth attribute of the seed. We regard the depth attribute of the seeds as an influential factor in energy scheduling. After many verification experiments, we propose a mutation strategy called Generations-Based Mutation (GBMutaion for simplicity), that is, with the increasing the seed depth, we assign more energy to the seed for the subsequent seed mutation.

Considering the adaptability of seed populations, we also add a fallback mechanism in the GBMutation strategy. That is to say, when the seeds in the current generation are still unable to find a crash after executing for a certain period, we would think that the functions or values that these seeds should have in this generation do not meet expectations. At this time, we will discard all the seeds in the current generation by reducing the seed depth and re-select the seed from the queue. From another perspective, the fallback mechanism of the GBMutation strategy can also prevent seeds from getting “stuck” and replace the test case in time.

Besides, we construct a model of the GBMutaion strategy as shown in Fig. 3. The GBMutaion strategy allows the optimal individuals of each generation to be retained and also ensures that the optimal individuals generated are not affected by those operations such as crossover and mutation to ensure the convergence of the algorithm.

We apply the GBMutaion strategy to the seed energy scheduling to ensure the better utilization of the seed value. Based on the GBMutaion strategy, we propose an energy scheduling model which is shown as Equ. 4.

$$p(i) = \begin{cases} \dfrac{2^{d(i)}}{log_2(f(i)+2)}, & d(i) < max_gene \\ \dfrac{2^{d(i)}}{log_2(f(i)+2) \cdot 2^{s(i)}}, & d(i) > max_gene \end{cases} \tag{4}$$

Where *p(i)* denotes the energy assigned to the current seed *i*; i.e., the number of the seed mutation, *max_gene* represents the maximum mutation generation; *d(i)* represents the depth of the seed; *s(i)* means that the number of times the seed input which executes a path is selected from the seed queue; *f(i)* means that the number of times the path of the seed has been executed in total, which also includes the number of times that it has been executed by other seeds. The increase of *s(i)* and *f(i)* will affect the evolutionary speed of the seed, so we put them on the denominator.

As shown in Equ. 4, our GBMutation-based energy scheduling strategy is very different from AFLFast's energy scheduling strategy (as shown in Equ. 5). Our strategy does not excessively discard seeds to ensure the diversity of the mutant seeds; and by introducing a fallback mechanism, our strategy also guarantees the convergence of fuzzing. Specifically:

On the one hand, through the monitoring of *f(i)*, our energy scheduling strategy still prefers to assign high energy to those seeds on the low-frequency path and low energy to those seeds on the high-frequency path. On the other hand, when *d(i)* is lower than *max_gene*, the energy allocated to the seed is increasing, but when *d(i)* is equal to *max_gene*, the seed energy value reaches the upper limit. To prevent the high-energy seed from being selected uniquely and continuously, we start monitoring by *s(i)*, and as the number of times the seed is selected from the queue increases, the energy allocated to the seed will begin to decline exponentially.

$$p(i) \propto \frac{s(i)}{f(i)} \tag{5}$$

4.4. Semantic information collection

The semantic information collection is one of the main parts of information integration. It refers to the DFI obtained through static analysis, mainly including the byte information and string information that is related to the *cmp* instructions and *cmp* functions. When using *cmp*, *cmpsb* and other *cmp* instructions in binary programs, or using *cmp* functions such as *strncmp* and *memcmp*, it is often the process of character or string matching. Especially in the face of such checks, it often leads AFL to go to erroneous fuzzing directions due to wrong judgments, seed selection errors, and insufficient seed mutation, which always makes the fuzzing process stuck. Therefore, to maintain the fast characteristics of AFL, we use static analysis to collect the sematic information and provide it to the fuzzer for oriented seed mutation.

For the semantic information at the instruction level, we collect the immediate data in the comparison instruction, mainly including single-byte information. As shown in Fig. 1, all the single-byte comparison information in each judgment instruction of module **(b)** and module **(c)** belongs to this type of information. Note that each basic block corresponds to a judgment instruction, and we use the immediate data in the comparison instruction to represent its corresponding basic block.

For the semantic information at the function level, there are some instructions (such as *mov, push*, etc.) used for setting function parameters before the function call instruction, and we would collect the string information existing in these instructions. Such semantic information mainly includes multi-byte information. As shown in Fig. 1, the string *"Hello"* in the basic block included in the module **(a)** is such information.

AFLPro collects single-byte and multi-bytes semantic information and uses this information for the implementation of seed mutations. At the same time, the collection of this information would, to some extent, solve the problem of the first type of error checking encountered by AFL in the seed selection stage, which is a guarantee that the selected seeds contain valid semantic information.

5. Evaluation

Through the above theoretical analysis, we conclude that AFLPro has the following advantages: 1) it maintains the fast, efficient, and stable characteristics of AFL; 2) by overcoming the blindness of fuzzing, it can bypass the sanity checks, thereby achieving a practical orientation of fuzzing. To further prove the effectiveness of AFLPro, we do comparison experiments with several state-of-the-art fuzzers such as AFLFast [8], VUzzer [6], AFLPlusPlus [11], etc. All comparative experiments demonstrate the advanced performance of AFLPro.

First, we use the DARPA CGC dataset to perform a separate validation for the energy allocation strategy in the seed mutation stage. On the one hand, referring to the time setting of the automated vulnerability mining competition, we set the evaluation time to be within 6 hours in the CGC evaluation experiment. On the other hand, to avoid the randomness of the experimental results, we performed multiple repeated experiments, which strengthened the reliability of the experimental data by presenting the average of the experimental results.

Secondly, for the overall verification of AFLPro, we chose the LAVA-M dataset and several real-world programs to complete the experimental evaluation. Among them, we set a 24-hour experimental evaluation span for the real-world programs to reduce the randomness of the experimental results, thereby providing definite proof for the effectiveness of AFLPro's vulnerability detection.

Furthermore, we verified the necessity of instrumentation optimization through experiments, and the experimental analysis results demonstrated that instrumentation optimization could be used as a research direction for future fuzzing improvement.

Our experimental environment is a virtual machine with an Ubuntu 16.04 system equipped with a 64-bit 4-core CPU and 4GB RAM. But for the real-world programs, in order to meet the installation requirements of our comparison tool VUzzer and ensure the consistency of the experimental environment, our experiments are performed on a virtual machine with a Ubuntu 14.04 system equipped with a 32-bit 4-core CPU and 4GB RAM.

5.1. Experiments on CGC dataset

To make the energy scheduling strategy proposed in this paper more convincing, this paper verifies its effectiveness based on the DARPA CGC dataset. The Challenge Binaries (CBs) are custom-made programs specifically designed to contain vulnerabilities that represent a wide variety of crashing software flaws. They are more than simple test cases, they approximate real software with enough complexity to stress both manual and automated vulnerability discovery [12].

Moreover, To ensure sufficient experimental support for the GBMutaion-based energy scheduling strategy of AFLPro, this paper controls a single variable. That is, the AFLPro energy scheduling model is separately integrated into the AFL, which is represented by *GBM(AFLPro)*, and our implementation *GBM(AFLPro)* is used for comparative experiments. AFLFast is an improved tool superior to AFL, so based on the CGC dataset, this paper only compares the experimental data of *GBM(AFLPro)* and AFLFast.

We obtained a total of 149 CGC binaries, of which 137 binaries were successfully tested. The specific reasons why the other 12 binaries were not successfully tested are described in Steelix [13], and we will not repeat them in this paper. Except for the output folder, we use the same execution command for *GBM(AFLPro)* and AFLFast. Besides, regarding the rules of the CGC competition and other popular automated vulnerability detection competitions, we conducted comparative experiments with a limited time of 1 h and 6 hours. We use the metric of the number of binaries that be triggered to crash. The various comparison results obtained by the experiment are shown in the Table 1. Among them, column 1 represents the experimental test time; column 2 and column3 represent the binary quantity that *GBM(AFLPro)* and AFLFast can trigger the program to crash, respectively; in column 4, for all CGC programs, >0 means that *GBM(AFLPro)* performs better than AFLFast (i.e., the number of binaries that can be triggered to crash is more), and <0 means that AFLFast performs better than *GBM(AFLPro)*; column 5 indicates the number of binaries that can be triggered to crash by *GBM(AFLPro)* but not by AFLFast; column 6 indicates the number of binaries that can be triggered to crash by AFLFast but not by *GBM(AFLPro)*; column 7 indicates the number of binaries that neither *GBM(AFLPro)* nor AFLFast can trigger them to crash. The experimental results of the Table 1 show that the combination of AFL and our proposed GBMutaion-based energy scheduling strategy can have better test performance than AFLFast, which also proves the effectiveness of our GBMutaion energy scheduling strategy.

In addition to counting the number of binaries that are triggered to crash, we also want to measure the number of crashes that trigger binary to crash. It is more powerful to prove the robustness and effectiveness of our energy scheduling strategy from these two aspects. Due to the space limitation, from the binary programs where both AFLFast and *GBM(AFLPro)* can trigger them to crash, we selected five CGC binaries randomly to show their experimental comparison results. As shown in the Table 2, column 2 and 3 present the number of crashes obtained by AFLFast and *GBM(AFLPro)* respectively during the fuzzing within 1 h. The remark part of the last column is a supplement to our experimental data. We use parallelized fuzzing in the experiment of CGC dataset, "-M" means deterministic seed mutation strategy during fuzzing, and "-S" means a non-deterministic seed mutation strategy during fuzzing. Our CGC-based experiment takes 1 h as a time limit. Taking remark of *CADET_00003* experimental data as an example, *avg(20*4(-S))* indicates that we use four parallel fuzzer processes to perform non-deterministic fuzzing, and 4 parallel fuzzers perform 1 h as 1 round. We conduct 20 rounds of fuzzing and calculate the average after summing all the experimental data. The average is our final experimental result. Similarly, *avg(20(-M))* means that we use 1 parallel fuzzer to perform deterministic fuzzing. Taking 1 h as 1 round, we perform 20 rounds of fuzzing and calculate the average after summing all the experimental data. The average is our final experimental result. Therefore, the supplementary expla-

Table 1
Statistics: the number of CGC binaries are crashed by *GBM(AFLPro)* or AFLFast.

	GBM(AFLPro)	AFLFast	GBM-AFLFast		GBM\AFLFast	AFLFast\GBM	Not Found
1h	23	13	>0	17	10	0	113
			<0	4			
6h	32	25	>0	19	9	3	102
			<0	13			

Table 2
Comparison: the number of crashes found.

Program	AFLFast	GBM(AFLPro)	remark
CADET_00003	33/91	75/180	avg(20*4(-S))/avg(20(-M))
CROMU_00046	75	80	avg(10*4(-S))
CROMU_00065	185	193	avg(10*4(-S))
CROMU_00087	1	4	avg(10*4(-S))
CROMU_00071	24/94	27/97	avg(10(-M))/avg(10*4(-S))

nation in our remark is to explain the origin of the experimental data in Table 2. After many experiments, the average experimental result is more convincing.

The experimental results in Table 2 prove that the ability of *GBM(AFLPro)* in detecting vulnerability is better than AFLFast. Besides, it is worth mentioning that although the measure of comparison between the fuzzers is the number of crashes obtained during fuzzing, we find that the quality of the crashes we got is higher than AFLFast after conducting the exploit experiment.

Our exploit experiment is done with the rex tool. Rex tries to exploit the crashes generated by AFLFast and *GBM(AFLPro)* respectively, and finally gets the conversion rate from crash to vulnerability exploit (i.e., POV, Proof of vulnerability). The experimental results indicate that the vulnerability exploitation conversion rate of *GBM(AFLPro)* is about 10% higher than AFLFast. Taking *CROMU_00071* as an example, *GBM(AFLPro)* has a vulnerability exploitation conversion rate of 26.8%, while AFLFast's conversion rate is 15.8%. In terms of vulnerability exploitation conversion rate, *GBM(AFLPro)* is 11% higher than AFLFast, which proves that *GBM(AFLPro)* gets a higher crash quality than AFLFast.

5.2. Experiments on LAVA-M dataset

Since the introduction of the LAVA-M dataset, the LAVA-M dataset has become a benchmark for researchers to test the performance of fuzzing tools. LAVA-M consists of 4 Linux utilities – *base64, who, uniq*, and *md5sum* – each injected with multiple bugs. And each injected bug has a unique ID differentiating with other bugs. To verify the advantages of AFLPro compared to other fuzzing tools, this paper continues to conduct experimental comparisons based on the experimental data provided in LAVA [14], VUzzer [6], InsFuzz [7] and other papers. We have reviewed the latest research materials and found many improved tools based on AFL. VUzzer and InsFuzz should be two state-of-the-art AFL-based tools. So we use them as experimental comparison objects to introduce the advantages and features of AFLPro. In addition, we have learned that the team of Yan proposes a program conversion method to improve the fuzzing, and implements a tool called T-Fuzz [4]. The idea of T-Fuzz is somewhat similar to that of Driller. That is, the AFL is used to generate input, and when AFL gets "stuck", it takes action. AFLPro also compares and analyzes with T-Fuzz based on the experimental results given in the T-Fuzz paper.

Like other fuzzing tools, AFLPro also tests each LAVA-M program with a limited time of 5 hours, and we don't turn on parallelized processes, i.e., don't use the "-M" and "-S" options. The specific experimental results are shown in Table 3. The results of AFLPro are shown in the last column and results from other fuzzers are shown in other columns. From the experimental results of *base64, uniq* and *md5sum*, we can see that AFLPro find more bugs than all other fuzzers listed in Table 3, and far more bugs in *who* than all fuzzers listed in Table 3 apart from InsFuzz. Besides, AFLPro finds some new bugs that the author of LAVA-M does not list. The new bugs found by AFLPro are shown in Table 4. We analyze the reasons why AFLPro found more bugs than VUzzer, InsFuzz, and T-Fuzz, and why AFLPro found fewer bugs than InsFuzz in *who* as below.

Comparison with VUzzer. Compared with VUzzer, the advantages of AFLPro are shown in three aspects:

1. AFLPro collects not only single-byte information in the *cmp* instruction during the static analysis phase, but also multi-bytes semantic information in the target program;
2. AFLPro puts forward BBA solution on the weight calculation of basic blocks, which better guides the direction of fuzzing;
3. AFLPro considers not only the localized basic block weights when making seed selection on the tuple, but also the weight of the seeds from a global perspective. Through both local and global considerations, AFLPro achieves better orientation than VUzzer and other fuzzing tools.

Comparison with T-Fuzz. Compared with T-Fuzz, AFLPro has better performance in automated fuzzing based on LAVA-M dataset. Note that, in the T-Fuzz column shown in Table 3, the number of crashes in brackets is the result of both manual and automated analysis. Besides, the overall design idea of T-Fuzz is similar to Driller, so we analyze that T-Fuzz has a common problem with Driller, that is, the global fuzzing direction is dominated by fuzzing, so what can be optimized and corrected by both two tools is the local fuzzing orientation problem. So T-Fuzz, like Driller, still has limitations in implementing direction sensitive fuzzing.

Another problem with T-Fuzz is that its scalability is not high. When the target program becomes larger or more complicated, it will have a "transform explosion" problem, which limits the execution speed of fuzzing.

Table 3
The number of bugs found by each fuzzer on LAVA-M DATASET.

Program	LAVA-M	FUZZER	SES	AFL-lafintel	Steelix	AFL-QEMU	AFL-Dyninst	VUzzer	T-Fuzz	InsFuzz	AFLPro
uniq	28	7	0	24	7	0	0	27	23(26)	11	**29**
base64	44	7	9	28	43	0	0	17	40(43)	48	**52**
md5sum	57	2	0	0	28	0	0	0	34(49)	38	**43**
who	2136	0	18	2	194	0	0	50	55(63)	802	**260**

Table 4
New bugs found by AFLPro.

Program	Unlisted Bugs	Total
uniq	227	1
base64	0, 2, 4, 6, 526, 527, 798, 804, 813	9
md5sum	281, 287	2
who	12, 16, 20, 24, 117, 125	6
who_p	※	54

Note:
We use who_p represents a parallelized experiment with running 8 AFLPro fuzzers on *who*. Because of two much unlisted bugs found by who_p, we use ※ to represent these bugs which are listed as follows.
※: 512, 4224, 2, 4, 6, 8, 522, 3083, 12, 514, 1038, 16, 1944,3082, 20, 24, 1049, 218, 1953,1936, 165, 298, 1071, 177, 307,1461, 312, 57, 4026, 59, 61,501, 1728, 197, 454, 459, 334,463, 336, 1361, 1892, 346, 477,481, 1026, 355, 1350, 488, 1388,1904, 117, 63, 125, 4223

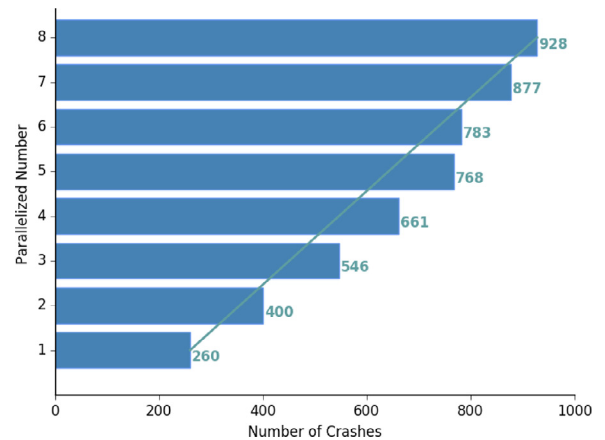


Fig. 4. Result of parallel experiments on *who*.

Comparison with InsFuzz. Compared with InsFuzz, AFLPro outperforms InsFuzz in testing *uniq, base64*, and *md5sum*. Especially in the test of *uniq*, the testing effect of InsFuzz is obviously behind AFLPro. Since InsFuzz does not open source, after analysis, we suspect that the reason for this is probably due to the inaccurate knowledge information collected by InsFuzz. The reason why we have such a guess is that there is often misleading information near the location where the comparison information exists in the binary program. Moreover, in the process of dynamic instrumentation, we cannot accurately analyze whether the comparison information is correct. When the collected comparison information is biased, it is easy to cause the wrong fuzzing direction. However, InsFuzz claims that it guides the random mutation based on the accurate knowledge set, which contradicts the fact that the comparative information is not always accurate. Therefore, we believe that the idea or method of such fuzzing should be further analyzed and verified.

Moreover, another advantage of AFLPro exceeds InsFuzz is that, apart from the static analysis and AFL tools, the implementation of AFLPro no longer depends on other tools, while InsFuzz is implemented with many tools. As mentioned in the paper of InsFuzz, it uses two instrumentation tools. One is the QEMU [15] integrated in AFL, which is to instrument *j** instructions and *call* instructions. Another instrumentation tool used by InsFuzz is the Dyninst tool [16,17], which instrument instructions of the *cmp* class to collect comparison information.

For the target program *who*, the reason why AFLPro performs worse than InsFuzz is the time-limit problem. Since there are thousands of sanity checks in *who*, this is rare in most of our target programs, and AFLPro is still mutating the only seed in the initial seed queue during the 5 h test period. A complete round of mutations based on the initial seed is not completed, i.e., the first round of seed mutation has not yet ended.

So we perform a parallelized experiment on *who*, as shown in Fig. 4, we find that when using parallelized fuzzing, the number of crashes that AFLPro can detect the crash is increasing within 5 hours.

In summary, the overall performance of AFLPro on the LAVA-M dataset is superior to the existing state-of-the-art fuzzing tools in terms of the number of bug found. It is particularly worth mentioning that for *md5sum, base64*, and *uniq*, the percentage of bugs detect by AFLPro within 1 h is 86%, 96%, and 100% of the number of bugs detected within 5 hours, respectively. And this is a stable result of many experiments, which is enough to prove the fast and stable advantages of AFLPro in vulnerability detection.

5.3. Experiments on real world programs

In order to measure the effectiveness of AFLPro in real life, we evaluate the experimentation in 4 real-world programs: *gif2png, pdf2svg, tcpdump, tcptrace* against AFL, VUzzer and AFLPlusPlus in the ability to find the unique crashes. AFLPlusPlus [11] is the latest proposed tool in the AFL family. AFLPlusPlus is the state-of-art, and its comparison experiments with AFLPro can show the absolute effectiveness of AFLPro.

It is worth noting that, these programs are also used to evaluate VUzzer but are different from the evaluation experiment of VUzzer. In VUzzer's evaluation experiment, it not only needs to perform static analysis on the target binary program but also needs to perform static analysis on the dynamic link libraries on which the target program depends. Moreover, it is also required to specify the entry addresses of the dynamic-link libraries during dynamic testing. Although VUzzer detects a lot of crashes, most of the crashes are not located in the target binary itself. Moreover, in terms of scalability, when fuzzing relies more on other information rather than the target program itself, its scalability will be greatly lim-

Table 5
32-bit system: The real-world programs evaluation results.

Program	Argument	Version	Unique Crashes		
			VUzzer	AFL	AFLPro
gif2png		2.5.8	8	75	86
pdf2svg	@@ output_page%d.svg all	0.2.2-1	0	2	4
tcpdump	-r @@ -nnvvvSeXX	4.9.2	0	0	0
tcptrace	-n @@	6.6.7	1	78	111

Table 6
64-bit system: The real-world programs evaluation results.

Program	Argument	Version	Unique Crashes	
			AFLPlusPlus	AFLPro
gif2png		2.5.8	28	62
pdf2svg	@@ output_page%d.svg all	0.2.2-1	3	4
tcpdump	-r @@ -nnvvvSeXX	4.9.2	0	0
tcptrace	-n @@	6.6.7	202	370

ited. The goal of this paper is to detect as much as possible the vulnerabilities of the binary itself, without paying attention to the vulnerabilities in the dynamic-link libraries that the target binary depends on. Therefore, in the process of dynamic fuzzing, this paper performs the static analysis only on the target binary and does not use the dynamic-link library as the test target, which also ensures the scalability of AFLPro.

Note that, we do not compare AFLPro with other fuzzers such as InsFuzz and T-Fuzz because their source code or binary tools are not released. Besides, the reason why we do not select *mpg321* and *djpeg* used in VUzzer's evaluation experiment as the target program are as follows.

- We need to customize the Ubuntu 14.04 virtual machine environment according to the installation requirements of the VUzzer and transplant AFL and AFLPro to the virtual machine environment for our evaluation experiment. But for some reason, we can't use *mpg321* correctly in this virtual machine. For the sake of the correctness of the VUzzer test, we do not choose *mpg321* as the test program.
- In the paper of VUzzer, it is mentioned that neither VUzzer nor AFL detects any related crash of *djpeg*. When getting rid of the dynamic-link library and only pay attention to the crash of *djpeg* itself, it not only makes the detection difficult, but we think the meaning of this binary program used in evaluation experiments is also greatly reduced, so we also do not choose *djpeg* as the test program.

In this paper, we conduct evaluation experiments by controlling a single variable, allowing VUzzer and AFLPro to perform the static analysis only on the target binary. VUzzer, AFL, and AFLPro are fed with the same input. In this experiment, we set a time limit of 24 hours for fuzzing to avoid any issues caused by randomness [18], and we do not use the "-M" or "-S" option to perform the parallelized fuzzing. It is worth noting that, because VUzzer requires the experimental environment to be a 32-bit system, and we found that the compilation of the QEMU environment on which AFLPlusPlus relies requires a 64-bit system during the installation of AFLPlusPlus, we conducted comparative experiments on two systems. The experimental results are shown in Table 5 and Table 6, respectively.

Table 5 shows the results of unique crashes triggered by VUzzer, AFL, and AFLPro. The second column in Table 5 is the parameters we used in the fuzzing process of the three fuzzers, where @@ represents the current seed input. According to the data in Table 5, AFLPro performs better than VUzzer and AFL, which proves that its vulnerability detection capability is higher. Moreover, compared to the number of vulnerabilities detected in the VUzzer paper that rely on dynamic link library information, the number of vulnerabilities detected by VUzzer in the evaluation experiments in this paper is significantly reduced. This also proves from the side that when relying on dynamic link libraries for vulnerability detection, many of the vulnerabilities are not located in the target program itself but located in the dynamic link libraries.

Table 6 shows the experimental comparison results of AFLPlusPlus and AFLPro. The experimental programs and its corresponding version number and parameters used are the same as those in Table 5. The only difference is that Table 6 is an experiment on a 64-bit system. According to the data in Table 6, AFLPro performs better than AFLPlusPlus. This comparison proves AFLPro's ability in vulnerability detection and demonstrates the absolute effectiveness of our proposed direction sensitive fuzzing method.

5.4. Instrumentation performance analysis experiment

Through experimental analysis, we find that when AFL uses QEMU to collect instrumentation information, it will generate a lot of meaningless instrumentation overhead during each dynamic execution of the binary program. The specific reasons are as follows:

- If our target program is a statically compiled binary, each time AFL re-executes the target program, the entry address of the program execution starts from the *_start* function. We think that the instrumentation instructions between the address of the *_start* function and the entry address of the actual *main* function are meaningless because the effective instrumentation information used in the process of dynamic fuzzing starts from the entry address of the *main* function.
- If our target program is a dynamically compiled binary, AFL will first dynamically link to the dynamic link libraries when it first executes the binary. Then, apart from the instrumentation between the *_start* function and the *main* function, the useless instrumentation cost of a dynamically compiled binary also in-

Table 7
Instrumentation performance analysis results.

Binary	Useless Insrumentation	Remark
Statically Compiled	46%	each execution
Dynamically Compiled	92%	first execution
	12%	non-first execution

cludes a large number of instrumentation loads generated during the dynamic linking process.

We use some statically compiled binaries of 670-700KB size and some dynamically compiled binaries of 50-60KB size as test programs. Based on the prototype AFL, the optimization performance analysis experiment is carried out on the instrumentation implementation. Our test results show that there is much space for performance optimization in the instrumentation implementation of AFL. Specifically, please see the data in Table 7 for details.

- For statically compiled binary programs, the data we obtained from experimental statistics is that about 46% of the instrumentation is meaningless.
- For the dynamically compiled binary programs, the statistical data we obtained is: For the first execution of the program, about 92% of meaningless instrumentation is generated, of which about 91% of the instrumentation is caused by dynamic linking, and about 1% of the instrumentation is between the *_start* function and the *main* function; For each subsequent program execution other than the first time, the instrumentation cost of the dynamic linking is no longer generated, but the instrumentation overhead from the *_start* function to the *main* function occupies about 12% in the total instrumentation.

As can be seen from the above experimental statistics, a single execution of the binary program would bring a lot of meaningless instrumentation overhead. The result of the uninterrupted execution of the binary program during the AFL fuzzing is that it accumulates more meaningless instrumentation cost, which would lead to a significant waste of computer resources. Therefore, this paper considers that it is necessary and meaningful to optimize the instrumentation implemented in AFL. The initial idea of our optimization is straightforward, that is, locating the entry address of the *main* function in the binary program and only the part after the address can be instrumented each time AFL executes the binary program. In this way, the instrumentation cost in the AFL fuzzing can be reduced, and the fuzzing speed can be improved. At present, in the AFLPro implemented in this paper, we have not made improvements and optimizations based on the implementation of the instrumentation in the AFL. But based on the support of the current analysis data, we consider in-depth research and implementation of this part in the future work.

6. Related work

Fuzzing is widely recognized by academics and industry for its advantages of fastness, efficiency, and stability. However, because of its blindness and randomness, fuzzing often deviates from the correct test direction when facing sanity checks in target program. To this end, the researchers have proposed several improvement methods in the research directions described below:

6.1. Coverage-Oriented fuzzing

Coverage-based fuzzing uses code coverage to measure the effectiveness of fuzzing. Usually, the path to an abnormal location in the program is challenging to detect, and the seeds provided by some traditional fuzzing tools are mainly executed on many invalid high-frequency paths. The execution of high-frequency paths in fuzzing limits the growth of code coverage. To improve the code coverage, AFLFast assigns high energy to the seeds on the low-frequency path while assigning low energy to the seeds on the high-frequency path. AFLFast's energy distribution strategy, combined with non-stop seed mutation, allows the fuzzing to trigger as many low-frequency paths as possible, thus enabling the fuzzing direction towards the low-frequency path. It is instructive for AFLFast to assign high energy to the seeds on the low-frequency path. Therefore, when the low-frequency path is detected in this paper, the same idea is adopted to allocate high energy.

AFLGo [19] takes the accessibility of the program's abnormal location as an optimization problem, and uses a meta-heuristic method to schedule and utilize the test seeds that are shorter than the abnormal locations in the target program. The seed distance calculation method used by AFLGo always directs the fuzzing to the shortest path that may reach the abnormal position (note that AFLGo's target objects are open source binaries), but this way of avoiding the longer path may cause the fuzzing to fail to reveal bugs hidden in the longer path. But AFLGo also provides us with an important thought, that is, the direction sensitivity problem of fuzzing can be transformed into a reachability problem for an optimal solution.

CollAFL [20] solves the problem of path collision caused by hash calculation, which, to some extent, corrects the problem that AFL deviates from the correct test direction in fuzzing due to path collision, and improves the accuracy of fuzzing. The above three fuzzing tools are the same as AFLPro. They are all coverage-oriented fuzzing tools, and they have improved and optimized AFL from different perspectives.

But in the aspect of the orientation of fuzzing, for the fuzzing tools or methods that rely only on coverage, it is difficult for them to pass sanity checks in the target program, such as the magic bytes check. Therefore, apart from the coverage, this article also introduces other metrics for passing sanity checks in the target program, such as basic block weights, seed weights, and so on.

6.2. Symbolic execution-Assisted fuzzing

Symbolic execution uses the symbol value instead of the actual value to execute the target program, it collects the path constraints and then generates an input that can pass sanity checks through the constraint solver. In theory, 100% code coverage can be achieved using symbolic execution. However, due to the existence of the path explosion, and the path constraint is too complicated so that the constraint solver cannot solve, which limits the scalability of symbol execution. Therefore, researchers use symbolic execution as an aid to fuzzing to improve the fuzzing performance, and this idea is used in the work of DART [21], SAGE [22], SYMFuzz [23], and Driller [3].

Driller is one of the most representative tools. Its main idea is to use fuzzing as the main method, let the symbolic execution perform the auxiliary test. When the fuzzing gets "stuck", Driller will call the symbolic execution to bypass the check, and then continue the fuzzing. However, the symbolic execution-assisted fuzzing represented by Driller is locally oriented, and the global direction still dominated by the fuzzing. Therefore, if the fuzzing deviates from the correct direction at the beginning, then the test result would be deviated and wrong. Besides, since symbolic execution is a

heavy-weight analysis technique, the stress of the constraint solving in symbolic execution is very large for longer paths or more complicated path constraints, so it is difficult to apply it to reality. Therefore, this paper does not choose the method of symbolic execution to assist fuzzing but uses the static analysis-based fuzzing.

6.3. Static analysis-Based fuzzing

Static analysis reasons about a program without executing it, and can achieve high code coverage of the target program. But due to the lack of dynamic information when the program is running, static analysis is not as accurate as fuzzing in vulnerability detection. Therefore, the vulnerability detection method combining static analysis with fuzzing can integrate resources and have complementary advantages of static analysis and fuzzing.

VUzzer uses static analysis to focus on the magic byte checks that exist in the *cmp* instruction compared to the immediate values and uses static analysis to calculate the weight of the basic block to guide the direction of the fuzzing. However, VUzzer has a certain degree of limitation on its execution speed and resource overhead due to the use of heavy-weight taint analysis. In this paper, when collecting information based on static analysis, for passing sanity checks, we are no longer limited to the magic byte check of the immediate value comparison in the *cmp* instruction. We also collect the string information corresponding to the parameters in the comparison function. Therefore, AFLPro can both pass magic byte checks and string comparison. On the other hand, for basic block weight calculation, this paper also proposes a BBA solution, using the thinking that more in line with the human to guide the direction of fuzzing.

Hawkeye [24] is an improved method based on AFLGo that guides the dynamic execution of fuzzing by collecting static information and achieves better results than AFL and AFLGo. However, the target object of Hawkeye is the binary program with source code, so its practical application is limited. In contrast, the target of this paper is the binary program without source code, so it has a wider range of practical applications.

InsFuzz [7] is an implementation that focuses on using static analysis to assist in passing sanity checks during fuzzing. But its dynamic way of determining key bytes depends on the extra instrumentation tool Dyninst, which not only brings more instrumentation overhead but also brings possibly incorrect byte information, which would lead to the wrong test direction.

6.4. Program transformation-Based fuzzing

Program transformation is the process of removing the detected sanity checks in the target program. Researchers have proposed some ways to bypass the complex sanity checks using the idea of program transformation. For example, TaintScope [5], MutaGen [25], AFL-lafintel [26,27], and T-Fuzz [4] are all fuzzing tools based on program transformation. But the implementation of TaintScope and MutaGen requires a lot of manual analysis, AFL-lafintel only works when you can access the source code of the program. T-Fuzz may generate false positives, and it would seek help from symbolic execution to filter out false positives. Therefore T-Fuzz is limited by symbolic execution, and it also has the problem of limited scalability. An important reason for improving AFL in this paper is to maintain AFL's scalability to the large-scale programs, as well as the advantages of fast and stable, so this paper does not adopt any methods or strategies related to symbolic execution.

7. Conclusion and future work

Fuzzing is acknowledged as a popular vulnerability detection technique in academia and industry. However, due to the limitations of its blind mutation, fuzzing cannot be effectively directed and is often difficult to pass sanity checks in the target program, which is relevant to the fuzzing direction carefully. Researchers have proposed many improved methods or tools for conducting fuzzing, such as Driller, VUzzer, etc. However, these methods still have some limitations, such as less scalability and less efficiency, as well as localized orientation and other issues.

This paper implements the direction sensitive prototype fuzzing method AFLPro based on AFL and focuses on seed selection. Around the seed selection, this paper designed the BBA solution and applied it to our proposed multi-dimensional oriented seed selection strategy to achieve fine-grained seed selection. Moreover, for other stages associated with seed selection, we have also proposed improvements. For example, we propose a GBMutation-based seed energy scheduling strategy and use static analysis for information integration. The application of all these strategies or methods enhances the orientation in fuzzing. We implement our fuzzing strategies in AFLPro and verify the effective orientation of AFLPro through comparative experiments. That is, on 92% applications of all datasets that we test in this paper, AFLPro performs better and has more possibility than the state-of-the-art fuzzers towards the crash locations in the target program.

However, AFLPro still has some limitations in practical applications, that is, for those programs that use code obfuscation and packing techniques, static analysis can not implement effective analysis for them. Due to the limitations of its static analysis module, AFLPro is limited in achieving effective vulnerability detection for such programs currently. Therefore, we will invest more research on how to eliminate the limitation of AFLPro while still ensuring its good features in the future. Furthermore, this paper proves the feasibility of instrumentation optimization by analyzing the instrumentation performance of AFL. As a future research work, we will conduct in-depth research on AFL's instrumentation optimization.

All in all, we will continue to explore more possibilities in automated vulnerability detection. For example, security is important and even critical for many applications of sensor networks, such as military and homeland security applications [28–32]. And not only in the field of sensor networks, but also in router networks [33], network services and protocols [34–36], internet of vehicles [37–39], internet of things[40,41], digital forensics [42], and other fields, there are many possible vulnerabilities to be exploited. Rapid vulnerability detection and recovery for the first time is essential to minimize the negative impact [34]. While in fact, automated vulnerability detection is not widely used in these fields at present. So in the future, to achieve automated vulnerability detection in these areas requires us to invest more human and material resources for in-depth research.

Declaration of Competing Interest

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

CRediT authorship contribution statement

Tiantian Ji: Conceptualization, Methodology, Software, Validation, Writing - original draft. **Zhongru Wang:** Conceptualization, Formal analysis, Writing - review & editing, Visualization, Funding acquisition. **Zhihong Tian:** Methodology, Writing - review & editing, Project administration. **Binxing Fang:** Writing - review & editing, Supervision. **Qiang Ruan:** Resources, Data curation, Writing - review & editing. **Haichen Wang:** Investigation, Writing - review & editing. **Wei Shi:** Writing - review & editing.

Acknowledgements

This work is supported by the Key-Area Research and Development Program of Guangdong Province (Grant No. 2019B010137004 and 2019B010136003), the National Key Research and Development Plan (Grant No. 2018YFB0803504 and 2019YFA0706404), the Major Scientific Research Project of Zhejiang Lab (No.2019DH0ZX01), and the BUPT Excellent Ph.D. Students Foundation (Grant No. CX2019115).

Supplementary material

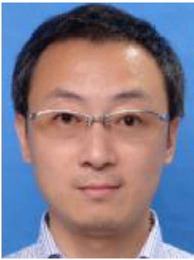
Supplementary material associated with this article can be found, in the online version, at 10.1016/j.jisa.2020.102497

References

[1] Tian Z, Shi W, Wang Y, Zhu C, Du X, Su S, et al. Real-time lateral movement detection based on evidence reasoning network for edge computing environment. IEEE Trans Ind Inf 2019;15(7):4285–94. doi:10.1109/TII.2019.2907754.
[2] lcamtuf. american fuzzy lop. 2019. http://lcamtuf.coredump.cx/afl/.
[3] Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, et al. Driller: augmenting fuzzing through selective symbolic execution.. In: NDSS, 16; 2016. p. 1–16. doi:10.14722/ndss.2016.23368.
[4] Peng H, Shoshitaishvili Y, Payer M. T-fuzz: fuzzing by program transformation. In: 2018 IEEE symposium on security and privacy (SP). IEEE; 2018. p. 697–710. doi:10.1109/SP.2018.00056.
[5] Wang T, Wei T, Gu G, Zou W. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: 2010 IEEE symposium on security and privacy. IEEE; 2010. p. 497–512. doi:10.1109/SP.2010.37.
[6] Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H. Vuzzer: application-aware evolutionary fuzzing.. In: NDSS, 17; 2017. p. 1–14. doi:10.14722/ndss.2017.23404.
[7] Zhang H, Zhou A, Jia P, Liu L, Ma J, Liu L. Insfuzz: fuzzing binaries with location sensitivity. IEEE Access 2019;7:22434–44. doi:10.1109/ACCESS.2019.2894178.
[8] Böhme M, Pham V-T, Roychoudhury A. Coverage-based greybox fuzzing as markov chain. IEEE Trans Softw Eng 2017. doi:10.1145/2976749.2978428.
[9] Hex-Rays. Idapython. 2019. https://www.hex-rays.com/products/ida/support/idapython_docs/.
[10] Shoshitaishvili Y, Wang R, Salls C, Stephens N, Polino M, Dutcher A, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE symposium on security and privacy (SP). IEEE; 2016. p. 138–57. doi:10.1109/SP.2016.17.
[11] vanhauser thc. american fuzzy lop plus. 2019. https://github.com/vanhauser-thc/AFLplusplus.
[12] Jiang Z, Feng C, Tang C. An exploitability analysis technique for binary vulnerability based on automatic exception suppression. Secur Commun Netw 2018;2018. doi:10.1155/2018/4610320.
[13] Li Y, Chen B, Chandramohan M, Lin S-W, Liu Y, Tiu A. Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering. ACM; 2017. p. 627–37. doi:10.1145/3106237.3106295.
[14] Dolan-Gavitt B, Hulin P, Kirda E, Leek T, Mambretti A, Robertson W, et al. Lava: Large-scale automated vulnerability addition. In: 2016 IEEE symposium on security and privacy (SP). IEEE; 2016. p. 110–21. doi:10.1109/SP.2016.15.
[15] Bellard F. Qemu, a fast and portable dynamic translator.. In: USENIX annual technical conference, FREENIX Track, 41; 2005. p. 46.
[16] Paradyn. Dyninst api. 2019a. https://www.dyninst.org/dyninst.
[17] Paradyn. dyninst. 2019b. https://github.com/dyninst/dyninst.
[18] Klees G, Ruef A, Cooper B, Wei S, Hicks M. Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. ACM; 2018. p. 2123–38. doi:10.1145/3243734.3243804.
[19] Böhme M, Pham V-T, Nguyen M-D, Roychoudhury A. Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security. ACM; 2017. p. 2329–44. doi:10.1145/3133956.3134020.
[20] Gan S, Zhang C, Qin X, Tu X, Li K, Pei Z, et al. Collafl: Path sensitive fuzzing. In: 2018 IEEE symposium on security and privacy (SP). IEEE; 2018. p. 679–96. doi:10.1109/SP.2018.00040.
[21] Godefroid P, Klarlund N, Sen K. Dart: directed automated random testing. In: ACM sigplan notices, 40. ACM; 2005. p. 213–23. doi:10.1145/1064978.1065036.
[22] Godefroid P, Levin MY, Molnar DA, et al. Automated whitebox fuzz testing.. In: NDSS, 8; 2008. p. 151–66.
[23] Cha SK, Woo M, Brumley D. Program-adaptive mutational fuzzing. In: 2015 IEEE symposium on security and privacy. IEEE; 2015. p. 725–41. doi:10.1109/SP.2015.50.
[24] Chen H, Xue Y, Li Y, Chen B, Xie X, Wu X, et al. Hawkeye: towards a desired directed grey-box fuzzer. In: Proceedings of the 2018 ACM SIGSAC conference on Computer and Communications Security. ACM; 2018. p. 2095–108. doi:10.1145/3243734.3243849.
[25] Kargén U, Shahmehri N. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering. ACM; 2015. p. 782–92. doi:10.1145/2786805.2786844.
[26] lafintel. laf-intel: Circumventing fuzzing roadblocks with compiler transformations. 2019a. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/.
[27] lafintel. Laf llvm passes. 2019b. https://gitlab.com/laf-intel/laf-llvm-pass/tree/master.
[28] Xiao Y, Rayi VK, Sun B, Du X, Hu F, Galloway M. A survey of key management schemes in wireless sensor networks. Comput Commun 2007;30(11–12):2314–41. doi:10.1016/j.comcom.2007.04.009.
[29] Du X, Xiao Y, Guizani M, Chen H-H. An effective key management scheme for heterogeneous sensor networks. Ad Hoc Netw 2007;5(1):24–34. doi:10.1016/j.adhoc.2006.05.012.
[30] Du X, Chen H-H. Security in wireless sensor networks. IEEE Wireless Commun 2008;15(4):60–6. doi:10.1109/MWC.2008.4599222.
[31] Du X, Guizani M, Xiao Y, Chen HH. Transactions papers a routing-driven elliptic curve cryptography based key management scheme for heterogeneous sensor networks. IEEE Trans Wireless Commun 2011;02(05):1223–9. doi:10.1109/TWC.2009.060598.
[32] Li M, Sun Y, Lu H, Maharjan S, Tian Z. Deep reinforcement learning for partially observable data poisoning attack in crowdsensing systems. IEEE Internet Things J 2019. doi:10.1109/JIOT.2019.2962914.
[33] Tian Z, Su S, Shi W, Du X, Guizani M, Yu X. A data-driven method for future internet route decision modeling. Future Generat Comput Syst 2019;95:212–20. doi:10.1016/j.future.2018.12.054.
[34] Xiao Y, Du X, Zhang J, Hu F, Guizani S. Internet protocol television (iptv): the killer application for the next-generation internet. IEEE Commun Mag 2007;45(11):126–34. doi:10.1109/MCOM.2007.4378332.
[35] Tan Q, Gao Y, Shi J, Wang X, Fang B, Tian ZH. Towards a comprehensive insight into the eclipse attacks of tor hidden services. IEEE Internet Things J 2018. doi:10.1109/JIOT.2018.2846624.
[36] Tan Q, Gao Y, Shi J, Wang X, Fang B, Tian Z. Toward a comprehensive insight into the eclipse attacks of tor hidden services. IEEE Internet Things J 2018;6(2):1584–93. doi:10.1109/JIOT.2018.2846624.
[37] Tian Z, Gao X, Su S, Qiu J, Du X, Guizani M. Evaluating reputation management schemes of internet of vehicles based on evolutionary game theory. arXiv preprint arXiv:190204667 2019. doi:10.1109/TVT.2019.2910217.
[38] Tian Z, Gao X, Su S, Qiu J. Vcash: a novel reputation framework for identifying denial of traffic service in internet of connected vehicles. IEEE Internet Things J 2019. doi:10.1109/JIOT.2019.2951620.
[39] Qiu J, Du L, Zhang D, Su S, Tian Z. Nei-tte: intelligent traffic time estimation based on fine-grained time derivation of road segments for smart city. IEEE Trans Ind Inf 2020;16(4):2659–66. doi:10.1109/TII.2019.2943906.
[40] Qiu J, Tian Z, Du C, Zuo Q, Su S, Fang B. A survey on access control in the age of internet of things. IEEE Internet Things J 2020. doi:10.1109/JIOT.2020.2969326.
[41] Tian Z, Luo C, Qiu J, Du X, Guizani M. A distributed deep learning system for web attack detection on edge devices. IEEE Trans Ind Inf 2019;16(3):1963–71. doi:10.1109/TII.2019.2938778.
[42] Tian Z, Li M, Qiu M, Sun Y, Su S. Block-def: a secure digital evidence framework using blockchain. Inf Sci (Ny) 2019;491:151–65. doi:10.1016/j.ins.2019.04.011.

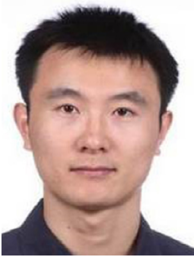
Tiantian Ji, Ph.D. candidate of cyberspace security with Beijing University of Posts and Telecommunications. Her current research interest is network security and information security.

Zhongru Wang, Ph.D. candidate of cyberspace security with Beijing University of Posts and Telecommunications. Senior Engineer of China Network Space Research Institute, Beijing, China. His current research interests include artificial intelligence and network security.



Zhihong Tian, received the Ph.D. degree in network and information security from the Harbin Institute of Technology, Harbin, China, in 2006. From 2003 to 2016, he was with the Harbin Institute of Technology, Harbin, China. He is currently a Professor, a Ph.D. supervisor, and the Dean of the Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou, China. His current research interests include computer networks and network security. Dr. Tian is the Standing Director of the CyberSecurity Association of China and a member of the China Computer Federation.

Binxing Fang, born in 1960. Professor and PhD supervisor. Academician of Chinese Academy of Engineering. His current research interests include computer architecture, computer network and information security.



Qiang Ruan, received the M.S. degree from Peking University. His current research interest include network security and artificial intelligence.

Haichen Wang, undergraduate student of Cyberspace security with Beijing University of Posts and Telecommunications, Beijing, China. His current research interests include Automatic Exploit Generation and Linux kernel security.

Wei Shi, received the bachelor's degree in computer engineering from the Harbin Institute of Technology, Harbin, China, in 2001, and the Ph.D. degree in computer science from Carleton University, Ottawa, ON, Canada, in 2007. She is an Assistant Professor with the University of Ontario Institute of Technology, Oshawa, ON, Canada. She is also an Adjunct Professor with Carleton University. Prior to her academic career, as a Software Developer and Project Manager, she was closely involved in the design and development of a large-scale electronic information system for the distribution of welfare benefits in China.