A Region-Sensitive Fuzzing Test Based on Multi-Objective Programming

Yongji Ouyang, Shuai Zeng, Yan Cao, and Qingxian Wang

Abstract—Fuzzing is an important technique for discovering vulnerabilities, unfortunately, it also offers fairly shallow coverage. To address these problems, this paper presents a region-sensitive fuzzing test based on multi-objective programming. Firstly, we perform region division on the test inputs through fine-grained taint analysis and offering mutated objects. Secondly, by combining the features of vulnerabilities, the attributes of functions and instructions were depicted for computing values of input regions’ attributes. Finally, this paper uses a multi-objective programming model to compute and rank the risk levels of these attributes, and the optimal one will be chosen to perform mutation. Experimental results show that the proposed approach can assist fuzzing test in choosing out and rank the risk levels of these attributes. Finally, this paper presents a region-sensitive fuzzing test based on multi-objective programming. In addition, invalid inputs are controlled within 12% and, and 74 vulnerabilities are found in real software.

Index Terms—Software vulnerability, fuzzing, multi-objective programming, region-sensitive.

I. INTRODUCTION

With the outbreak of "Heartbleed" and "Prism", people are increasingly concerned about the safety of personal privacy, therefore, how to eliminate vulnerabilities in software, now become more important. Wang [1], Rebert [2] and so on, as the representatives of researchers have made a lot of achievements, but there are still many places need to be improved. Fuzzing is an automated security test method for discovering software vulnerabilities by providing invalid or random input and monitoring the system under test for errors or potential vulnerabilities [3], it is not only able to discover vulnerabilities in traditional desktop software, such as zzuf [4], MiniFuzz and other tools, but also can discover vulnerabilities in the smart mobile devices, such as the achievements of Maji [5] and DroidFuzzer [6].

Initially, the methods of traditional fuzzing are widely used to detect the bug of code, however, as the construction method is too random, the lack of contact between the data and vulnerabilities, the defects of traditional fuzzing are also very obvious. To address the problems of lacking comprehension to target programs, researchers propose intelligent fuzzing which takes advantage of the intrinsic knowledge of programs as well as the information generated during the fuzzing process to assist test, and it can discovery vulnerabilities more effectively. The main challenge of this approach is how to generate effective input samples to cover and trigger vulnerabilities. To solve this challenge, a fuzzing approach based on symbolic execution, typically represented by KLEE [7], SAGE [8] and dowser [9], is gradually becoming a hot topic in research. This approach records an actual run of the program under test on an initial input, collects execution paths and generates corresponding path constraints which are then negated one by one to get new constraints, and at last produces new inputs by solving these new constraints which can effectively increase coverage. However, the most representative open-source prototype tools such as Fuzzgrind [10], avalanche [11] and KLEE still cannot be effectively applied in test large real binary applications because both symbolic execution and constraint solving need vast computational resources.

In addition, although researchers develop some fuzzing tools based on thoughts of evolution, represented by EFS [12], the basic idea of these tools is still to observe runtime behaviors of programs, and then mutate program inputs using rules of evolution strategy like genetic algorithm, not considering the features of vulnerabilities. It is still less effective, according to researcher’s observation, particularly at string manipulation programs. Moreover, technologies based on template description mutation, represented by Peach [12], are also limited to test some specific applications and lacks general utilization, despite the fact that they can parse specific file formats, get through initial inspection on programs effectively, and thus cover deeper paths to perform test.

To increase efficiency of fuzzing and reinforce mutation operation, this paper used a multi-objective programming [13] and proposed a region-sensitive fuzzing test based on this model, which can assist fuzzing test in choosing out and mutating input sections that are most probable to trigger exceptions. The essential contribution of this paper is threefold: First, it performs region division on test inputs through fine-grained taint analysis and provides mutation objects for fuzzing test; Second, it accurately depicts input attributes of functions and instructions that affects input regions and provides conditions for computing risk levels of input regions; Lastly, it utilizes multi-objective programming model to assess and rank the risk levels of input regions so as to provide basis of test priorities of input regions for fuzzing.
II. OVERVIEW

The vigorous development of information technology and Internet industry brings software engineering into a new phase, whereas, followed by the sharply rising difficulty of software test. In general, nowadays software safety test mainly encounter the following several challenges: firstly, data interaction in existing software is very numerous and jumbled, and multiple types of external inputs can be accepted; secondly, due to the much larger scale and higher construction complexity of software, causes of software errors are also becoming more complex, such as the vulnerability CVE-2009-2629 in Nginx; thirdly, aiming at complex and diverse error code areas, it is difficult to choose out the object codes under test. Fig. 1 shows a program execution flow that contains vulnerabilities and in which lots of objects are processed, multiple inputs are accepted and a certain position may be affected by multiple inputs. Meanwhile, there may exist or not vulnerabilities in one execution flow and the goal of test is to find paths to vulnerabilities from input positions and then specially construct inputs to trigger corresponding program exceptions. Thus it is a very difficult work to rapidly and accurately find vulnerabilities in programs.

However, as more and more vulnerabilities are discovered, researchers find that positions where vulnerabilities are triggered have some common characteristics. They generally gathered together at hot spots, such as code areas with high cyclomatic complexity and areas that contain dangerous operation functions (strcat, strcpy, etc) [14]. Fig. 2 shows the errors in two examples of vulnerability CVE-2014-1761 and CVE-2011-0978. It can be seen that in CVE-2014-1761 an error occurs when the function memcp y is called because the objective index pointer is not checked, although there is a checking condition in CVE-2011-0978 when the array index is used, this condition can still be bypassed and a corresponding overflow occurs. After the further analysis, both copied contents and checking conditions in the two examples are affected by program inputs. Thus, if the input scope that affects them can be correctly identified, exceptions can be triggered more rapidly.

To increase the discovery efficiency of vulnerabilities, this paper proposes a region-sensitive fuzzing test based on multi-objective programming, and designed a prototype fuzzing test tool, whose general framework is shown in Fig. 3.

As is shown above, this test framework is mainly divided into three phases. The phase of region division divides the input data into fine-grained semantic units of input region as the mutated objects; the phase of multi-objective programming ranks the risk of input region through the establishment of planning rules and providing test basis for the fuzzing test, which mainly includes the two aspects of multi-objective programming model and the set of region attribute functions; at last, the optimal input field is selected to generate new inputs and realize the test. The following sections give the details of the key algorithms and techniques that the approach in this paper involves.

III. INPUT REGION DIVISION

This section mainly researches the division approach through which the input data can be divided into multiple regions so as to provide the optimal mutation units. For illustration purposes, a definition is firstly given as below.

Definition 1(region): the input data that meets a specific format is comprised of several regions each of which is a unit that has a specific semantic meaning and consists of sequential input bytes.

Thus, the key to granularity division on input regions is to determine the borders of each region. The idea adopted in this paper is as this: a region is a semantic processing unit which is regarded as a whole when the program runs calculations on it during the procedure of data processing. So, the data in the same region tends to be co-processed by functions or instructions. When one region contains multiple bytes, these bytes needs to be used together so as to form an integral semantic unit provided for instructions of arithmetic operation, compare or other types. In this way, according to the processing granularity of functions or instructions, input data can be divided into different semantic units. Besides, since regions are independent of each other, it seldom occurs that one function parameter or an instruction access contents belonging to different semantic regions. Based on the above reasons, this paper design the input region algorithm as shown in Algorithm 1.

Algorithm 1: input region division
Input: the execution trace T and the divided function or instruction set A
Output: the division set $\mathcal{X}$ of input regions

(1) Init $S, S', \mathcal{X} = \emptyset$
(2) foreach $\alpha$ in $A$
(3) $S = \{\alpha\}$
(4) foreach stmt in reverse($T.statements$)
(5) If $\{stmt.lhs\} \cap S' = \emptyset$ then
According to the requirement, we need to know the degree of danger level. In fact, as the size of input file is controllable, their input offsets, and then performs interaction on these offsets, which can largely increase the efficiency of input region division.

Intuitively, utilizing this algorithm to acquire input regions for testing owns the below advantages: 1) the granularity of division is more close to the format of input data, which can easily get through the format-checking of programs; 2) the granularity of division reflects the information of the program’s processing on input data, thus avoids the blindness of black box test. In addition, it requires a bit of explanation that different from reverse parsing of data formats, the main goal of this algorithm is to determine the borders of regions so as to provide fine-grained test objects for fuzzing test without regarding the concrete semantic information of regions.

IV. MULTI-OBJECTIVE PROGRAMMING OF INPUT REGION

After getting the input region scope of test sample, what this paper care is how to correctly select the sizes and orders of input regions to perform mutation and then generate new samples to go on test. To solve this problem, this paper uses multi-objective programming to assist fuzzer in choosing.

A. Multi-Objective Programming Modeling

As we know, in traditional fuzzing test, there exists a certain degree of arbitrariness in choosing the mutation offset which ranges from one bit to all the input size and of which the input regions got in section 4 are subsets. Provided that $X$ is a solution set of input regions, decision variable $x = (x_1, x_2, \cdots, x_n) \in X$ is one solution of $X$, and meanwhile the object functions of attributes of $x$ are represented as $f_1(x), f_2(x), \cdots, f_n(x)$, therefore, for a given $x \in X$, each value of the attributes $f_1, f_2, \cdots, f_n$ can be calculated. According to the requirement, we need to know the degree of danger of the input region, so as to guide the fuzzer with danger level. In fact, as the size of input file is controllable, this paper might as well assume that all the constraints of $x$ can be expressed by inequalities, as in (1).

\[
g_i(x) \leq 0, i = 1, 2, \cdots, m
\]  

(1)

So the set of all feasible solutions can be represented by (2).

\[
X = \{x \in R^N | g_i(x) \leq 0, i = 1, 2, \cdots, m\}
\]  

(2)

Then the objective equation of the input region as shown in (3), $f_1, f_2, \cdots, f_n$ represents the attribute functions of input region and $h(f_1(x), f_2(x), \cdots, f_n(x))$ represents the sum of all dangerous property values, this formula could guide us choose the input regions with higher probability vulnerability trigger.

\[
\begin{align*}
\text{Max} & \ h(f_1(x), f_2(x), \cdots, f_n(x)) \\
X & = \{x \in R^N | g_i(x) \leq 0, i = 1, 2, \cdots, m\}
\end{align*}
\]  

(3)

The following sections will concretely introduce how to choose the attribute functions of input regions for getting the most dangerous input regions.

B. Input Region Attributes

As a portion of the program inputs, during the program’s processing on inputs, intuitively, the input regions can make influences on the functions and instructions in the program; meanwhile vulnerabilities are closely tied to functions and instruction, and for instance, the cause of buffer overflow generally is that the dangerous library functions such as strcpy and memcpy are called. Thus this paper decides to choose functions and instructions as the attributes of input regions. Of course, the input regions can contain other attributes like the complexity of data formats of the input regions. For simplifying the model, this paper only considers the two aspects including functions and instructions.

1) Function attribute

As to the applications ready to be tested, this paper can classify the functions of them into two classes. One is system library functions and the other is sub-functions that belong to the modules of the applications themselves. Thus, this paper depicts the function attributes based on these two classes.

a) Subfunction danger attribute

As to the danger attributes of the sub-functions in applications, this paper utilizes Cyclomatic Complexity to measure them [15]. Cyclomatic complexity is one approach of software measurement which can be applied to measure the complexity of programs. It adopts the main idea: the larger the number of independent paths in programs, the more complex the programs. Therefore, the higher the cyclomatic complexity of functions, the higher complexity of CFG, it have a higher probability of error in them. Provided that the CFG of a function is $G$, the cyclomatic complexity can be calculated by (4), where $V(G)$ is the cyclomatic complexity of this function, and $E$ is the number of the edges in the CFG while $N$ is a basic block(BBL), and $P$ represents the connecting part which is 1 because the CFG is interconnected.
Subfunctions, the equation of danger attribute of subfunctions in an input region can be represented, as in (5), where \( G_i \) indicates the \( i \)-th subfunction affected by the input region \( I \).

\[ DSFunc(I) = \sum_{i=1}^{n} V(G_i) \]  

**b) Library function danger attribute**

According to statistics, the vulnerabilities are triggered mainly because of the incorrectly calling dangerous functions which mainly refer to that some print and output classes and string functions in the standard library functions of C or C++ do not perform boundary check when implemented, while programmers frequently use these functions when they are developing software. If these functions are not properly called and boundary check is not performed, some safety problems may be caused. We list parts of common dangerous functions in Table I.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Insecurity description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcpy, wcscpy, lstrncpy, strcat, wcscat, strncpy, ...</td>
<td>write tainted data into memory, like rep movsd, stos, and so on.</td>
</tr>
<tr>
<td>functions similar to printf (printf, _sprint, ...)</td>
<td>buffer overflow</td>
</tr>
</tbody>
</table>

In the practical test, more and more guided test approaches that aim at dangerous functions were introduced [9], [16]. Based on these research findings, in general, the more dangerous library functions affected by an input region, the higher degree of danger the library functions have. Thus, this paper utilizes the number of dangerous library functions affected by an input region to measure the danger attribute of related library functions in this region. Equation (6) is the representation of the library function danger attribute.

\[ DLib(I) = \sum_{i=1}^{n} |isCallUnsecurity_i| \]  

where “is Call Un-security” indicates whether the call is a dangerous function call.

**2) Instruction attribute**

Similar to the function attributes, instructions attributes are divided into two classes. One is the instruction hot attribute and the other is the instruction danger attribute. These two situations will be discussed in the following sections.

**a) Instruction hot attribute**

When dealing with the input data, the procedures and complexity of processing each input region are different. In general, the more instructions in one region are processed, the higher access frequency of this region, so test this region can affect more instructions and accordingly the covered code area will be larger. Thus, this paper measures the instruction hot attribute based on the number of instructions processed in the related input region, as in (7), where \( I \) indicates the input region, \( taintSrc(ins) \) is the input source of instruction \( ins \), and \( N \), which can be adjusted according to the actual conditions, generally represents the mean value of the number of instructions contained in the function.

\[ InsHot(I) = |\text{Ins} | \text{/} N, \text{where } \text{Ins} = \{\text{ins} | taintSrc(ins) \cap I \neq \emptyset\} \]  

**b) Instruction danger attribute**

Similar to calls of dangerous functions, the system dangerous instructions refers to those that may cause exception due to unsafely processing external data. According to the causes of vulnerabilities in recent years, dangerous instructions can be classified as the following several classes:

1) Instructions that write tainted data into memory, such as mov [ebx], edx, and stos, and so on.
2) Instructions that cause the register EFLAG to be tainted, such as conditional jump instructions of ja, jna, jb, and so on.
3) Loop memory copying instructions that make the count tainted, like rep movsd which makes the copying length ecx tainted.
4) Math operation instructions with tainted data. Many integer overflows are due to the unsafe calculations on external data and there are some common math operations of this kind, such as mul, inc, sub, add, and so on.

If this paper sets the set of above instructions as \( D \), the expression of the instruction danger attribute of one input region can be calculated by (8).

\[ InsD(I) = |\text{Ins}(\text{ins} \in D) \lor (taintSrc(ins) \cap I \neq \emptyset)\]  

**C. Calculating the Rank of Input Regions**

After acquiring the attributes of input regions, the final goal of this paper is to rank these input regions based on their degree of risk so as to provide the basis of mutation priority for fuzzing. Since the attributes of each input region exist in several aspects, we cannot simply accumulate the results of equations from (5) to (8). In general, it is needed to perform weight setting on each attribute, then, according to the results of tests, we should update weights value constantly, the final objective equation as shown in (9).

\[ \max_{x \in X} DSFunc(x) + DLib(x) + InsHot(x) + InsD(x) \]

| \( X = \{x \in \mathbb{R}^N | g_i(x) \leq 0, i = 1, 2, \ldots, m\} \) |
| \( \omega = (\omega_1, \omega_2, \ldots, \omega_m), \sum_{i=1}^{m} \omega_i = 1, \omega_i \geq 0 \) |

According to the above section, the influencing factors of the objective equation mainly have four functions, therefore, we use the artificial neural network model [17] to calculate the risk value of the input region, as shown in Fig.4. Meanwhile, in the process constantly adjust the weight value of each function, makes the attribute weights in accordance with the current test process.
Finally, based on the calculated level of the input region, select the input region descending order for data generation, and then begin to test.

V. EVALUATION

This section shows the effectiveness and correctness of the proposed approach. First, this paper chooses a real world application to illustrate the tutorial and feasibility of the proposed approach. Finally, three experiments were evaluated for how well it discover the vulnerabilities known and unknown in the real world software, as well as advantages and disadvantages compare to other methods.

A. Experiment Environment

The proposed method is implemented named RFuzz which based on Pin [18], and the evaluation is conducted on a well-known commodity computer which is equipped with 4GB of main memory, 3.4 GHz dual-core CPU and Windows XP SP3. All evaluations are performed by running some real world software scenario.

B. Instance Analysis

To verify the feasibility of the approach in this paper, it chooses a drawing tool (MSPaing) as a study object and the bmp file shown in Fig. 5 as an input, and takes them for an example to illustrate the concrete implementation procedures of each key step.

As mentioned in this paper, it is firstly needed to perform taint analysis on test objects and record the related information, and then perform input region division. Via RFuzz, this paper can get the relationship of function calls and the corresponding accessed input regions, as shown in Fig. 6.

The Fig. 6 provides some important data regarding data access, this paper can know what functions access what sections of input regions and the relationship of function calls as well based on it. For instance, DIBWidth(char *) : <18, 21> represents that the four sequential bytes starting from the offset Ox18 in input data are accessed by one certain instruction in the function DIBWidth. It can also been seen that the accessed input regions during the procedures of function calls reflect the functional information of input regions. For example, the input region <28,29> goes through three function calls: CreateImg->DIBNumColors->DIBBitsPixel, and according to the format of BMP, the sequential two bytes starting from the offset Ox2c represents the bits of every pixel, which conforms to the functional description of the function DIBBitsPixel. Thus, if the mutation is properly performed in appropriate input region scope, the semantics of input format can be maintained and the test can be guaranteed to penetrate deep inside the tested program.

After the determination of input regions, the next job is to compute out the value of each attribute of every input region. Table II exhibits the information of the four attributes of several acquired input regions. In the Table II, R represents the region, A represents the attribute, S represents the danger attribute of subfunction, L represents the danger attribute of library function, H represents the hot attribute of instruction, and D represents the danger attribute.

After the acquirement of attributes in Table II, RFuzz will utilize the multi-objective programming to compute out the value of each attribute of every input region, then perform priority ranking on input regions according to these values, and at last choose out the optimal input region to begin the test.

C. Verifying the Test Effectiveness

To verify the effectiveness of the test approach proposed in this paper, three experiments are designed, where the first one is to test the ability of discovering the pre-existing vulnerabilities, the second is to test the ability of discovering unknown exceptions, and the last is to compare RFuzz with other tools.

1) Experiment I

To verify the capability of the approach in this paper on the
detection of pre-existing vulnerabilities, this paper selects five published vulnerabilities in recent years, rewrites the corresponding Pocs as normal test samples, and then perform the input region division and ranking on input data. To intuitively measure the effect of detection, this paper names the bytes which trigger vulnerabilities as vulnerable bytes and define the ranking-priority ratio of the input region as the quotient of its ranking divided by the total sum of input regions. The smaller the ratio is, the higher accuracy of input region will become. For instance, if the ranking-priority ratio of one region is 8%, then the region ranks in the top 8% of all ranked input regions. Table III shows the concrete test results, where $S$ is the size of test sample, $N$ is the number of Input region that were divided, $W$ is the flag whether the vulnerable bytes were divided, $R$ is the Rank of vulnerable bytes, $P$ is the ranking-priority ratio of one region, and $T$ is the test time.

<table>
<thead>
<tr>
<th>CVE number</th>
<th>$S$ (Byte)</th>
<th>$N$</th>
<th>$W$</th>
<th>$R$</th>
<th>$P$(%)</th>
<th>$T$ (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2010-3333</td>
<td>1276</td>
<td>134</td>
<td>√</td>
<td>12</td>
<td>8.9</td>
<td>13</td>
</tr>
<tr>
<td>CVE-2010-4371</td>
<td>458</td>
<td>83</td>
<td>√</td>
<td>11</td>
<td>13.5</td>
<td>7</td>
</tr>
<tr>
<td>CVE-2011-0978</td>
<td>2341</td>
<td>207</td>
<td>√</td>
<td>18</td>
<td>8.7</td>
<td>12</td>
</tr>
<tr>
<td>CVE-2013-3934</td>
<td>18489</td>
<td>581</td>
<td>√</td>
<td>35</td>
<td>6.0</td>
<td>21</td>
</tr>
<tr>
<td>CVE-2014-1761</td>
<td>25491</td>
<td>615</td>
<td>√</td>
<td>43</td>
<td>7.0</td>
<td>27</td>
</tr>
</tbody>
</table>

As show in the Table III, the approach in this paper can correctly recognize the bytes that trigger vulnerabilities, and in the divided input regions, vulnerability bytes generally rank in the top. The highest ranking is 11, the lowest ranking still reaches 43, and meanwhile the ranking-priority is 8.82% in average. This data indicates that the region-sensitive fuzzing test based on multi-objective programming in this paper can better preferentially choosing and test the input area which is more likely to trigger vulnerabilities. Also, in the process of test, we found that the convergence value of weight are about $(0.195, 0.391, 0.158, 0.256)$ and $(0.178, 0.435, 0.151, 0.236)$, and for WINWORD, Winamp, WinEXCEL and WPS respectively. If you want to simplify the model in a specified test, this value can be used as a reference.

2) Experiment II

To verify the discovery effect of the approach in this paper on unknown exceptions, this paper chooses an office software named WPS 2013, a video software named Baofeng Player 5 as test objects to carry on the experiment.

Since the system process data in units of a computer word, the mutation unit of input region in this paper is also a word. Provided the size of this word is $K$ (In 32-bit system $K=4$ and in 64-bit system $K=8$), as to the input region $P$, the below mutation is adopted:

1) If $|P|<K$, every single byte is randomly replaced.
2) If $|P|=K$, equal-length bytes (namely DWORD in 32-bit architecture) are replaced.
3) If $|P|>K$, random lengths of random characters are replaced.

Meanwhile, the following conditions are satisfied: as to each input region, the max times of test it is 400, which is to say that once the test on one region is ended, the test on another lower-level input region is started; the time of one test is set as 24 hours. Then the test results can be acquired as shown in Table IV, where $S$ is the size of input file, $N$ is the number of test samples, $T$ is total the number of invalid input, $P$ is the percentage of invalid samples and total samples, $E$ is the number of total exceptions, and $U$ is the number of unique exceptions.

<table>
<thead>
<tr>
<th>Program</th>
<th>$S$ (KB)</th>
<th>$N$</th>
<th>$T$</th>
<th>$P$(%)</th>
<th>$E$</th>
<th>$U$</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPS 2013</td>
<td>33</td>
<td>67216</td>
<td>7864</td>
<td>11.7</td>
<td>269</td>
<td>42</td>
</tr>
<tr>
<td>Baofeng Player 5</td>
<td>4</td>
<td>4219</td>
<td>443</td>
<td>10.5</td>
<td>81</td>
<td>7</td>
</tr>
<tr>
<td>SumatraPDF</td>
<td>82</td>
<td>58152</td>
<td>5500</td>
<td>9.4</td>
<td>185</td>
<td>25</td>
</tr>
</tbody>
</table>

In Table IV, the invalid inputs denote that because of the test cases’ dissatisfying the required format the tested software will prompt that the input data is not correct. It can be seen from the data in Table IV that the invalid inputs can be controlled within 12% of all RFuzz test cases, which illustrates that the input region division in this paper makes little destruction on the semantics and can assist the tester in entering into the inside of programs. However, when this paper utilize MiniFuzz to carry on tests running the same configuration, it find out that as to the above three software all the ratios of invalid inputs exceed 50%, from which it can see that the approach in this paper owns considerable advantages in maintaining semantics, and meanwhile in average 74 software exceptions are discovered by MiniFuzz within 24 hours, which indicates that RFuzz can also effectively discover the vulnerabilities in programs. Even so, this paper also finds that there exists a high exception repetition rate in the approach of this paper and in this aspect further perfection should be made.

3) Experiment III

This experiment still uses WPS 2013, SumatraPDF and Baofeng Player 5 as the objects of test. Meanwhile, Pingrind and MiniFuzz are chosen out as comparison tools to test these three software, and all of the tools test time is 24 hours. The test results are shown in Fig. 7.

![Fig. 7. The number of exceptions of MiniFuzz, Pingrind and RFuzz.](image)

As the results from Fig. 7, given the same test time, the efficiency of exception detection in this paper increases 10 times than that of the fuzzing approach used by MiniFuzz. And for the software SumatraPDF, RFuzz finds some exceptions that MiniFuzz cannot find in it, whereas Pingrind almost cannot find any bug. As for the tool Pingrind, it adopts an typical intelligent test approach based on dynamic symbolic execution, which imports the intrinsic knowledge of the object systems in the test process to assist the fuzzing test so as to generate test cases with more pertinence and uses some technological means such as intermediate representation, taint transmission analysis, symbolic execution, path constraint solving, and so on. Since the test on large applications needs numerous computing resources, to
proceed the test smoothly, this paper sets the max number of constraints respectively as 500 and 1000, and then separately use Pingrind to carry on tests, but cannot find any bug, from which it can be seen that as to larger applications the fuzzing test approach based symbolic execution owns limited effectiveness with limited test resources. As for the tool MiniFuzz, it is a fuzzing test tool used to simplify the deployment of fuzzing test and its main idea is to construct abundant testing data to test programs, paying attention to the model of randomized data but not analyzing the connection between data and the program itself and lacking feedback learning about the pre-existing testing data.

VI. CONCLUSION

From the perspective of software safety test, performing mutation on bytes with more pertinence can effectively increase the efficiency of fuzzing test. This paper firstly proposes a region-sensitive fuzzing test based on multi-objective programming which firstly perform region division on the test inputs through fine-grained taint analysis, then accurately depicts the attributes that are likely to trigger vulnerabilities, performs calculations on all attributes and rankings on all input regions via multi-objective programming, and at last according to the sort order perform mutation test on the preferred input region. It can be seen from the above theories and experiment data that the region-sensitive fuzzing test based on multi-objective programming proposed in this paper owns higher effectiveness than traditional fuzzing test and more practicability than the fuzzing test based on dynamic symbolic execution.

ACKNOWLEDGMENT

We would like to thank Qiang wei, Jianshang Peng, Zhongxu Yin, and our anonymous reviewer for their comments and suggestions.

REFERENCES


Yongji Ouyang received his B.S. and M.S. degrees in network engineering from Zhengzhou Information Science and Technology Institute (ISTI) in 2008 and 2011, respectively. He is currently a Ph.D. candidate in the State Key Laboratory of Mathematical Engineering and Advanced Computing at ISTI. Dr. Ouyang current research interests include network security, and security of mobile and desktop software.

Shuai Zeng received his B.S. and Ph.D. degrees from Beijing University of Posts and Telecommunication in 2009 and 2011, respectively. She is currently a research assistant in the State Key Laboratory of Management and Control for Complex System at Institute of Automation, Chinese Academy of Sciences. Dr. Zeng research interests include network security, mobile security, and social network.

Yan Cao received his B.S., M.S. and Ph.D. degrees in network engineering from Zhengzhou Information Science and Technology Institute (ISTI) in 2006, 2009 and 2013, respectively. He is currently an assistant professor in ISTI. Dr. Cao current research interests include network security, mobile security, and social network.

Qingxian Wang received his B.S. degree in the Department of Applied Mathematics from Zhengzhou Information Science and Technology Institute (ISTI) in 1982 and M.S. degree in the Department of Computer Science and Technology from Peking University. He is currently a professor in ISTI. Prof. Wang current research interests include network security, mobile security, trusted computing, malicious code analysis, and vulnerability discovery.