Research on Fuzz Testing Framework based on Concolic Execution

Xiong Xie and Yuhang Chen*

Department of Information, Dalian Maritime University, 116026 Dalian, China

ABSTRACT

Vulnerability discovery technology is a significant aspect of the current. The work of this paper is to design and realize a fuzz framework based on concolic execution using C++. This framework is composed of instrumentation module, path constraint generation module and solver module. To improve the efficiency, the traditional technology method was optimized. It avoids the problem of path explosion that when an external call occurs. The experimental results show that our framework can trigger vulnerabilities successfully and expand code coverage.

INTRODUCTION

With the rapid development of the Internet, information security is closely related to our daily life. We almost use online payment everyday, such as apply pay and wechat pay. If there are software vulnerabilities, it will result in property damage. The root of these problems is software vulnerabilities. So vulnerability discovery technology is a significant aspect.

At present, vulnerability discovery can be divided into static discovery and dynamic discovery, according to whether run software. Static discovery means to analyze the program through lexical, syntax, control flow analysis and scan the code to verify the quota such as safety, reliability and maintainability without running the software. Dynamic discovery is to achieve and analyze the information generated during the process of running software. The common methods of vulnerability discovery are: symbolic execution[1], reachable path analysis[2], taint analysis[3] and fuzz testing[4].

In this paper, a fuzz testing framework is designed and implemented. Section 2 introduces the state of the art on the related techniques. Section 3 introduces the detail and implementation of framework. Section 4 presents the results of the framework and compares with others. Section 5 presents conclusions and gives some future directions.
RELATE WORK

Common Methods

CONCOLIC EXECUTION

Concolic (a portmanteau of concrete and symbolic)\(^5\) is a software verification technique that performs symbolic execution. It is called dynamic symbolic. Symbolic execution is a form of static program analysis method.

Compared to the traditional symbolic execution, concolic adds specific run-time information in symbolic execution, so that it can avoid false positive. False positive is a warning that does not represent a real bug. After concolic testing, the framework uses the SMT solver\(^6\), because the solver ability is limited, the traditional one will get stuck and lose the path leading to false negative, which refers to a bug that is not reported. But concolic will use the current specific values instead of symbolic value to get a complete, simplified symbolic expressions containing the specific values. It makes the simplified path constraint back to the theory supported by SMT Solver, so that on the one hand reduces the restrictions on the use of SMT solvers, on the other hand to improve the code coverage.

SMT SOLVER

SMT solver is used after concolic execution. SMT (Satisfiability Module Theories) is the extension of SAT (The Satisfiability problem)\(^7\). SMT is better than SAT in the field of artificial intelligence and formalization methods, such as resource planning, timing reasoning, compiler optimization, and so on.

The common SMT solvers are Z3\(^8\), Yices, CVC3, STP\(^9\), etc. This article chooses Z3, Z3 is developed by Microsoft's SMT solver. It is one of the best solvers, it supports a variety of theories and mainly used in software validation and software analysis. Since Z3 has been widely recognized, the Z3 solver is used in this paper.

FUZZ TESTING

Fuzz testing is focused in this paper, it is a random testing technique based on error injection, which analyzes the program’s execution result and the state of testing program state by inputting test set to the software. This framework uses the idea of fuzz testing. It injects many test cases to trigger vulnerabilities. From 2002 till now, fuzz testing tools emerge in an endless stream, and they are oriented to different aspects. For example, the browser-oriented managlem\(^10\), the FileFuzz\(^11\] and SPIKEfile\(^12\] are designed for the format file. After that, the focus of fuzz testing shifts to the framework optimization. In 2007, fuzz testing combined others techniques, it can be carried out on the basis of a higher code coverage. Code coverage is also known as assertion coverage, which measures whether each branch of a function is executed. It is an important criterion in software testing.

Existence Tools

Present tools using concolic are DART\(^13\] and CUTE\(^14\]. In DART, an external call is expanded at the invoked place to execute the implementation test in an external call. After DART appeared, CUTE is proposed. It improves the DART handling of pointers, aliases, and other variable methods. But it does not change the
way to handle external calls. These calls will cause path explosion. So in this paper, in order to solve this problem function summary is used. In the solver module part in Section 3, some formulas are given to illustrate the reason why to use function summary. As time goes on, there are also some effective tools such as S2E[15], Driller and so on.

In the common tools use depth-first search or breadth-first search such as KLEE[16]. They can only generate one path for one time. In this paper, the framework use path traversal algorithm based on generation. Instead of generating one, our method generate more than one. The details of path traversal algorithm introduces in Section 3 and in section 4 there is a contrast experiment to show the result.

**SYSTEM FRAMEWORK**

A fuzz framework based on the concolic execution is proposed in this paper. In this framework, the external input of the test program mainly includes two categories: one is file, network, I/O function, and so on. The other one is the system environment variables such as system time. This framework generates new test cases by generating constraints from external inputs and uses SMT solvers to solve the constraints. It is composed of three main modules: the instrumentation module, path constraint generation module and the SMT solver module.

**Main Steps of the Framework**

Step one: Symbol flags for external input are generated through using the symbolic mark for the path that sensitive function passed. Sensitive function (also known as sensitive nodes) of the external input refers to the function operating on the file and memory operation, such as malloc(), strcpy(), fopen(), etc in C.

Step two: The path constraint set are calculated. Symbol execution tree are generated by symbols formed in step one. In the tree, a path means a branch from root node to a leaf node. New path constraint sets are generated via path traversal algorithm based on generation.

Step three: The nonlinear constraints in these constraints are performed using concolic, to make them become linear. The concolic uses actual value to replace nonlinear one. These constraints are solved using the Z3 solver to generate new input test data and then guide the program toward new path execution.

Step four: For next input round, code coverage are calculated by pin. The test case with higher code coverage is selected.

The system framework overview as shown in Figure 1.
**Instrumentation Module**

This module contains two parts: one is to find sensitive function by pin. The other one is to calculate code coverage. This module mainly introduces calculating code coverage.

The aim of using pin is in charge of recording each code, reading information, writing memory, the system function and the return value during execution. Whether the base code is executed or not can be determined by the return value of the inserted code, which inserted in pin. For the basic block of code that has been executed, it will be recorded in the executed basic block. After the program is finished, code block coverage is calculated by comparing the code recorded in the basic block with the total code.

**Path Constraint Generation Module**

This module contains two parts: the generate symbolic execution tree and calculate path constraint.

**GENERATE SYMBOLIC EXECUTION TREE**

The symbol execution tree is derived from the control flow chart (CFG). The framework converts the involved program input into a symbolic variable, expresses the program statement as a symbolic variable expression, and then converts the program into a symbol through the simulation of these symbolic variables executed in the CFG. In the symbolic execution tree, the block is represented as node, jump is represented as an edge. The path of a symbol execution tree from root to the leaf node is regard as the conditional executable path.

**CALCULATE PATH CONSTRAINT**

This module uses path traversal algorithm based on generation [17] (generational search algorithm). The core idea of this algorithm is: based on the current path constraints, the algorithm negates the target branch dependent input constraints in turn and generates multiple path constraints in a new structure.

In the Figure 2, left subtree means the original path, right subtree means path after negated. According steps mentioned before, the second generation path is $AB \rightarrow BF \rightarrow IL$, $AB \rightarrow BF \rightarrow FK$, $AH \rightarrow HM \rightarrow MN$. In the second generation, the path that negated $AB \rightarrow BF \rightarrow IL$, $AB \rightarrow BF \rightarrow FK$ has occurred before. So reverse the path $AH \rightarrow HM \rightarrow MN$ to get the third generation: $AH \rightarrow HM \rightarrow MO$. The all path generated by the algorithm are shown in TABLE I.
<table>
<thead>
<tr>
<th>First generation</th>
<th>Second generation</th>
<th>Third generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg AB \to AH&amp;HI&amp;IJ )</td>
<td>( AH&amp;HI&amp;\neg IJ \to AB&amp;BF&amp;IL )</td>
<td>( AH&amp;HM&amp;MO )</td>
</tr>
<tr>
<td>( AB&amp;\neg BC \to AB&amp;BF&amp;FG )</td>
<td>( AB&amp;BF&amp;\neg FG \to AB&amp;BF&amp;FK )</td>
<td>( AH&amp;\neg HI&amp;IJ \to AH&amp;HM&amp;MN )</td>
</tr>
<tr>
<td>( AB&amp;BC\neg CD \to AB&amp;BC&amp;CE )</td>
<td>( AH&amp;\neg HI&amp;IJ \to AH&amp;HM&amp;MN )</td>
<td>( AH&amp;HM&amp;MO )</td>
</tr>
</tbody>
</table>

The main process is in the Figure 2, the tree is a symbolic execution tree generated by test program. The node A, B, C... in tree means the program block. The path constraint (PC) of original path is \{AB&BC&CD\}. Firstly, the branch constraints in the original path constraints are negated partly, and the next generation (first generations) of new path constraint sets is generated: \{\neg AB\}, \{AB &\neg BC\}, \{AB &BC &\neg CD\}. Secondly, the first generation are repeated above the process to get the second generation, the third generation, etc.

Figure 2. Path traversal algorithm process.

**Solver Module**

This module contains two part: one is to confirm constraint by concolic based on function summary [18] and the other one is to generate test input by Z3 solver

**CONCOLIC BASED ON FUNCTION SUMMARY**

Concolic adds the actual execution of the program data into the testing process and follows the actual path, so that it avoids the path explosion problem of CFG in the process of symbolic execution partly. The path explosion is the core problem of symbolic execution, it means the exponential growth of path number due to the growth of path length. Especially when programs contain loops and recursion, the number of paths is infinite. To solve this problem we used a concept called function summary. In order to better describe the module, the following keywords definition used in this paper and formula are given.

Definition 1. External call: Given a function f, f is called by other function, this process is called external call.

Definition 2. Total path amount: It contains the program’s self path and the path in external call.
Definition 3  Test path amount.
Assume that the enter function $\alpha_n$ (the begin function called by program) of the tested program $P$ has $n-1$ external call functions. The external call function are expressed as $\alpha_1, \alpha_2, \ldots, \alpha_{n-1}$, the n-1 functions are the set $\{\alpha_1, \alpha_2, \ldots, \alpha_{n-1}\}$. These function are called for $m_1, m_2, \ldots, m_{n-1}$ times, the number is expressed as a set of $\{m_1, m_2, \ldots, m_{n-1}\}$. In this case, the test path amount $f'$ is calculated by formula (1)

$$f' = f_1 + f_2$$ (1)

In formula (1), $f_1$ is the path amount without external call. It is total path amount $\phi_n$ minus the sum of the external calls amount as formula (2)

$$f_1 = \phi_n - \sum_{i=1}^{n-1} m_i$$ (2)

$f_2$ is the sum of the number of paths $f(\alpha_i)$ ($1 \leq i \leq n-1$) that are called by the function itself multiplied by the number of calls $m_i$ ($1 \leq i \leq n-1$), as the formula (3)

$$f_2 = \sum_{i=1}^{n-1} f(\alpha_i)m_i$$ (3)

when the function $\alpha_i$ has been called for $m_i$ times, $\alpha_i$ will be retested for $m_i$ times. $f_2$ will tends to infinity, so we use function summary

Definition 5 function summary
An abstract to an external call is called a function summary. The summary of function $F$, $\mu_F$ is as the formula (4), in it $in_F$ as the conjunction of the input constraint of function $F$, $out_F$ as the conjunction of the output constraint of function $F$.

$$\mu_F = in_F \land out_F$$ (4)

When a program is being executed, a function summary is generated if an external call is encountered. During subsequent executions, when the call is encountered again, use the function summary instead, thus avoiding the concolic execution of the implementation again. In summary with the function of external summary instead of multiple calls to the function can effectively avoid the problem of path explosion.

INSTRUCTIONS OF Z3 SOLVER

A example of using Z3 solver is to solve $x - z = z + 5$. $\text{Z3_L_FALSE}$, $\text{Z3_L_TRUE}$, and $\text{Z3_L_TRUE}$ are the result of equality equation, the following meanings are false, true and undefined.

The steps of using Z3 solver are as follows. Firstly using Z3's API interface, these constants and variables are converted into data structure variables that are used internally by Z3. Secondly, according to the expression of the content, the corresponding interface function is called. Thirdly using the assert interface, the logical expression is generated by corresponding converted expression by Z3 and
call the check interface to solve. Finally the solution given by the solver is true. The solution of the example given above is $x = 9$, $z = \frac{5}{2}$.

**EXPERIMENT AND ANALYSIS**

**Algorithm Implementation**

By giving a binary tree first to the breath-first algorithm, and then based on the generation of traversal, the results can be seen that the number of the latter path is significantly larger than the former, more suitable for the framework of this article.

Generational search algorithm has been implemented in C++, the result are shown as Figure 4. Compared with it, the result of breadth-first algorithm are shown as Figure 3.

![Figure 3. Breadth-first algorithm result.](image1)

![Figure 4. Generational search algorithm result.](image2)

**Function Testing**

This program, a typical one in this field, is tested using the framework proposed in this paper,

```c
int main (int argc ,char * argv[]){
    int n= 0;
    char input[4];
    FILE * fp = fopen(argv[1],"r");
    fread(input,1,4,fp);
    switch(input[0])
    case:'b'n++;
    case:'a'n++;
    case:'d'n++;
    case:'!'n++;
    if (n > 2)  abort();// error
}
```

This program’s function is a read operation on the file. Firstly, it reads the specified file 4 bytes of content. The variable n is a count variable, and if $n > 2$, the exception is triggered. It is hard to trigger a vulnerability if the data generated by the 4 bytes is randomly generated.
There are \(2^{8 \times 4}\) possible combinations, but only 5 of these combinations can trigger this vulnerability. So the trigger probability is about \(1/2^{30}\), using the traditional fuzzing tests are very difficult to find vulnerability. By using method in this paper, first signifying the input to \(i_0,i_1,i_2,i_3\), the initial path constraint is \(\{i_0 \neq b, i_1 \neq a, i_2 \neq d, i_3 \neq !\}\), after negate it becomes \(\{i_0 = b, i_1 \neq a, i_2 \neq d, i_3 \neq !\}\), and then get the solution \(\{i_0 = b\}\).

Theoretically, the need for a 2 round of testing can trigger bug. The actual test results are shown in TABLE II.

**TABLE II. TEST RESULT.**

<table>
<thead>
<tr>
<th>Method</th>
<th>Original input</th>
<th>Test amount</th>
<th>Test file amount</th>
<th>Exception file amount</th>
<th>Code coverage(%)</th>
<th>Test time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our method</td>
<td>Good</td>
<td>8</td>
<td>78</td>
<td>1</td>
<td>12.5%</td>
<td>252s</td>
</tr>
<tr>
<td>Fuzz testing</td>
<td>Good</td>
<td>8</td>
<td>78</td>
<td>0</td>
<td>2.4%</td>
<td>15324s</td>
</tr>
</tbody>
</table>

After 8 rounds of testing, code coverage is 12.5%. Because of some system functions, so there is only 12.5%. But the main function in all the functions of the code have been tested, if the system function are not considered, only from the angle of function, the probability function code for testing 100%. And traditional fuzz does not trigger this program vulnerabilities, so code coverage rate is 0%

This paper also experiments with FolderChangesView, a software used to monitoring file reader and writer. Compare to FileFuzz, the result is as below. To test, many files are used for testing, the file suffix of them are .doc, .txt and so on. The result in the TABLE III shows our approach works better on code coverage.

**TABLE III. TEST RESULT.**

<table>
<thead>
<tr>
<th>Method</th>
<th>Test amount/B</th>
<th>Amount of test file</th>
<th>Code coverage(%)</th>
<th>Test time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our method</td>
<td>52510</td>
<td>7524</td>
<td>17.5%</td>
<td>789754s</td>
</tr>
<tr>
<td>FileFuzz</td>
<td>52510</td>
<td>7524</td>
<td>4.8%</td>
<td>15324s</td>
</tr>
</tbody>
</table>

CONCLUSIONS

In this paper, a fuzz testing framework based on concolic has been designed and implemented. In this framework, the common symbolic execution is replaced by concolic, and more accurate paths are generated by the realization of the path traversal algorithm based on generation. Test cases are generated to improve the code coverage by solver. In order to demonstrate the validity of the framework, there two experiments, algorithm implement and function testing. The next stage, the key point is to optimize continually the code coverage and increase efficiency of symbolic execution. The framework should be expanded powerfully enough for the complex software. The ultimate goal of the framework is to transform to the mobile phone terminal and it can also test effectively for the apps.
ACKNOWLEDGMENT

This work is supported by the Science and Technology Funds of Dalian (Grant No.2015A11GX010) and the Fundamental Research Funds for the Central Universities (Grant No.3132016308)

REFERENCES