Software Vulnerability Detection using Backward Trace Analysis and Symbolic Execution

Hongzhe Li, Taebeom Kim, Munkhbayar Bat-Erdene and Heejo Lee
Div. of Computer and Communication Engineering
Korea University
Seoul, South KOREA
(hongzhe, ktb88, munkhbayar, heejo)@korea.ac.kr

***Abstract*—Software vulnerability has long been considered an important threat to the safety of software systems. When source code is accessible, we can get much help from the information of source code to detect vulnerabilities. Static analysis has been used frequently to scan code for errors that cause security problems when source code is available. However, they often generate many false positives. Symbolic execution has also been proposed to detect vulnerabilities and has shown good performance in some researches. However, they are either ineffective in path exploration or could not scale well to large programs. During practical use, since most of paths are actually not related to security problems and software vulnerabilities are usually caused by the improper use of security-sensitive functions, the number of paths could be reduced by tracing sensitive data backwardly from security-sensitive functions so as to consider paths related to vulnerabilities only. What's more, in order to leave ourselves free from generating bug triggering test input, formal reasoning could be used by solving certain program conditions. In this research, we propose backward trace analysis and symbolic execution to detect vulnerabilities from source code. We first find out all the *hot spot* in source code file. Based on each *hot spot*, we construct a data flow tree so that we can get the possible execution traces. Afterwards, we do symbolic execution to generate program constraint(PC) and get security constraint(SC) from our predefined security requirements along each execution trace. A program constraint is a constraint imposed by program logic on program variables. A security constraint(SC) is a constraint on program variables that must be satisfied to ensure system security. Finally, this hot spot will be reported as a vulnerability if there is an assignment of values to program inputs which could satisfy PC but violates SC, in other words, satisfy $PC \wedge \overline{SC}$. We have implemented our approach and conducted experiments on test cases which we randomly choose from *Juliet Test Suites* provided by US National Security Agency(NSA). The results show that our approach achieves $Precision$ value of 83.33% , $Recall$ value of 90.90% and $F1\_Value$ of 86.95% which gains the best performance among competing tools. Moreover, our approach can efficiently mitigate path explosion problem in traditional symbolic execution.**

***Keywords*-symbolic execution, program constraint, vulnerability detection, static analysis.**

I. Introduction

Software security has long been considered an important issue due to the increasing number of attacks. In order to identifying software vulnerabilities effectively, software security testing has gained significant attention and become an active research area. In the past decade, many approaches have been proposed to detect software vulnerabilities [2] [3] [6] [9] [13].

Static analysis has been proposed to discover software vulnerabilities by analyzing source code or software binary. The advantage of this approach is the large coverage of code and the access to the internal structures or workings of a software or application. However they often approximate or even ignore runtime conditions which makes them suffer from high false positive rate.

Dynamic analysis monitors program execution to detect security flaws [4] [5] [7] [8] [10]. These tools try to trigger software vulnerabilities by giving test input data to a program and monitor its runtime behavior. When there is an abnormality detected during execution, there may be a vulnerability existed. For example, the program accessed a buffer outside its bounds. Although dynamic analysis reduces false alarm rates, it requires the generation of actual bug triggering test inputs which often make us cannot find critical security flaws in a reasonable time. What's more, the coverage of the whole huge test inputs space is either too much time costly or just impractical to achieve which will cause false negatives. Researchers recognized the problems and use dynamic symbolic execution to cover the test inputs space more efficiently.

Symbolic execution has been proposed to detect vulnerabilities and has shown good performance in some researches. However, they often cannot solve the problem of huge search space of program paths so that they are either ineffective in path exploration or do not scale well to large programs. What's more, most of dynamic symbolic execution works on binary level which means that they cannot get useful information from source code. SecTAC [1] works on source code level, however, they can only detect vulnerabilities from the paths they get from functional test cases which could make them to miss vulnerabilities from the paths their test cases cannot cover.

The number of paths could be reduced by tracing sensitive data backwardly from security-sensitive functions so that we only consider vulnerability related paths. Whats more, formal reasoning such as solving $PC \wedge \overline{SC}$ of the program conditions by doing symbolic execution could leave us free from test inputs generation which makes the whole detecting process more efficient. In this paper, we propose a novel method to detect software vulnerabilities from source code using backward trace analysis and symbolic execution. Different from traditional one direction analysis in symbolic execution, we backwardly analyze the execution traces and then performing symbolic execution forwardly. We first analyze the source code to find out all the hot spot which could possibly generate a vulnerability using pattern matching technique. A hot spot is defined as: a security-sensitive function with sensitive data among its arguments. Sensitive data is derived from untrusted input sources like input files, or network messages, command line options. Then, beginning with each hot spot, we backwardly analyze the source code and build the data flow tree of each hot spot. From the data flow tree, we could get the possible execution paths that could reach to this hot spot. Based on each execution trace, we do symbolic execution to get the program constraints(PC) and security constraints(SC). A program constraint is a constraint imposed by program logic on program variables. A security constraint(SC) is a constraint on program variables that must be satisfied to ensure system security. Finally, this hot spot will be reported as a vulnerability if there is an assignment of values to program variables which could satisfy PC but violates SC, in other words, satisfy $PC \wedge \overline{SC}$. We try to solve this constraint by using a proper constraint solver such us SMT solver. Our evaluation tells that our approach achieves $Precision$ value of 83.33% , $Recall$ value of 90.90% and $F1\_Value$ of 86.95% which gains the best performance among competing tools. Moreover, our approach can efficiently mitigate path explosion problem in traditional symbolic execution. Our contributions could be described like this:

- Different from other static analysis technique, we take program trace execution conditions into consideration to detect software vulnerabilities which could dramatically decrease the false positive rate.
- We mitigate the path exploration problem in symbolic execution by considering only vulnerability related paths. Tracing the sensitive data used in hot spot backwardly to recover the execution paths using data flow tree makes us only focus on those branches related to sensitive data. This makes the detection process more efficient
- The $PC \wedge \overline{SC}$ constraint solving makes formal reasoning applicable to accurately detect software vulnerabilities. Meanwhile, it makes us free from generating actual bug triggering test inputs

II. Related work

Our research has been conducted based on some previous knowledge about related works. There are several works about dynamic symbolic execution and test cases generation to find software flaws which take advantage of the combination of concrete and symbolic execution. CUTE [4] and DART [9] can automatically generate test cases to traverse different execution paths by the use of a combination of symbolic and concrete execution. They improve the classical symbolic execution by making a distinction between the concrete and the symbolic state of a program. The code is basically run unchanged, and only statements that related to the program input are treated differently.

EXE [17] and KLEE [18] are developed to automatically generate high-coverage test cases and to discover deep bugs and security vulnerabilities in a variety of complex code. They provide the speed necessary to quickly solve the constraints through a combination of low-level optimizations and a series of higher-level ones such as caching and irrelevant constraint elimination. However, the greatest challenge for those mechanisms is the scalability problem. We still cannot get a good answer to the question on how to handle the exponential number of paths. Hence, if a very tiny vulnerability exits in a program with millions of code statements, it will be very time consuming to find it out. Unlike these tools, our approach first identifies the potential vulnerable points by pattern matching technique and then uses backward program slicing to reduce the irrelevant paths that we need to analyze. This helps us to focus on vulnerability related paths only and quickly verify the potential vulnerable points.

Automated whitebox fuzzing, implemented as a tool named SAGE [16], is a recent method to do software security testing which extends the scope of systematic dynamic test generation from unit testing to whole application testing. It is able to scale to large file programs with millions of code lines and execution traces with billions of machine instructions. However, SAGE works on binary level so it cannot take advantage of useful information from source code.

There are also some static analysis tools such as CodeAuditor [15] implemented by Lei Wang which works on source code level. It analyzes the program using program slicing to extract the paths only relevant to the potential vulnerable point. They mitigate the path explosion problem in a smart way, however, they need to instrument the original source code to get the range of program variables which makes it time consuming. The closest work to ours is SecTAC [1] proposed by Dazhi Zhang. SecTAC works on source code level and it uses the test cases from traditional functional software testing to get the popular execution traces of the program.

By performing symbolic execution along each trace with satisfiability analysis, it can quickly and correctly detect vulnerabilities which locate on the traces covered by the test cases. The reuse of the test cases generated from previous functional software testing helps them to avoid the problem of path exploration. However, this makes it unable to discover the vulnerabilities which are not covered by the traces. In other words, they may miss serious vulnerabilities.

Our approach does not rely on any test cases. We take advantage of the common feature of the software security vulnerabilities to firstly identify all the potential vulnerable points in the whole program and then verify each potential vulnerable points known as hot spot to get the finally vulnerability report which makes us avoid the problem of false negatives.

III. Our approach

Discovery of vulnerabilities in a program is a key process to the development and management of secure systems. Nowadays, static analysis tools and methods cannot get rid of high positive rate. Dynamic analysis techniques such as Fuzzing depend on program input significantly which probably make us miss critical vulnerabilities located in some traces which are not covered by program input.

Another problem of these techniques is the coverage of huge input space which is usually a time consuming work. Some tools using dynamic symbolic execution are powerful to detect vulnerabilities.However, the path exploration problem makes them ineffective and the process of finding bug triggering test inputs usually puts a huge burden on testers.

However, we found out that during practical use, most of paths are actually not related to vulnerabilities and software vulnerabilities are usually caused by the improper use of security-sensitive functions.

As a remedy, we try to mitigate the path exploration problem by tracing sensitive data backwardly from security-sensitive functions so as to consider only paths related to vulnerabilities. Afterwards, vulnerabilities are reported by solving $PC \wedge \overline{SC}$ after doing symbolic execution on those paths without finding test inputs that actually trigger vulnerabilities.

Generally, Figure 1 gives us a schematic overview of our proposed approach. The process of our approach could be described as the following 5 steps.

1) **Hot spot detection**: A hot spot(known as sink in other related works)is defined as: a security-sensitive function with sensitive data among its arguments. Functions such as *strcpy,memcpy,printf* are sensitive to cause security problems.
   Sensitive data is derived from untrusted input sources like input files, network messages or command line options. We find all those hot spot(potential vulnerable) by simply using pattern matching strategy. This process could be finished in a short time.
2) **Data flow tree construction**: In order to eliminate the paths we need to analyze, we trace the sensitive data used as argument in hot spot backwardly to build a data flow tree. The tree is used for generating possible execution paths. Actually, during the process of constructing the tree, paths which are not related to hot spot or sensitive data have automatically been eliminated.
3) **Possible execution paths generation**: After the construction of data flow tree, possible execution paths are generated automatically from the data flow tree.
4) **Program constraints(PCs) and security constraints(SCs) generation**: Symbolic execution is performed on generated traces to get program constraints and security constraints.
5) PC **and** SC **verification**: A satisfiability checker is used to check if $PC \wedge \overline{SC}$ is satisfiable. We have found a vulnerability if the condition could be solved. The solution given by the satisfiability checker is then used to generate test data to uncover the vulnerability. In the following part, we discuss these steps in more detail and provide examples for better illustration.

A. Hot spot detection

A hot spot(known as sink in other related works)is defined as: a security-sensitive function with sensitive data among its arguments. Functions such as strcpy,memcpy,printf are sensitive to cause security problems. Sensitive data is derived from untrusted input sources like input files, or network messages, command line options.

Based on our study, vulnerabilities are usually caused by overflowed value used in sensitive functions. These sensitive points we called hot spot. We can classify hot spot according to the different features of sensitive functions

- **Memory copy:** The sensitive data is used as argument to be copied in a destination buffer(strcpy,memcpy). When destination buffer cannot hold the sensitive data, serious security problems may occur like buffer overflow
- **Memory access:** The sensitive data is used as an array index or a pointer offset, which may cause arbitrary bytes memory overwritten or read(e.g, an information leakage attack).
- **Memory allocation:** The sensitive data is used as argument in memory allocation functions(e.g., malloc, alloca) and it usually cause insufficient memory allocation.
- **Format string:** The sensitive data is used improperly as argument in format functions(e.g., printf, sprintf). Attacker can take use of this vulnerability to take control of the system.

We analyze the given source code to find all of potential vulnerable points which we called hot spot by simply using pattern matching strategy.

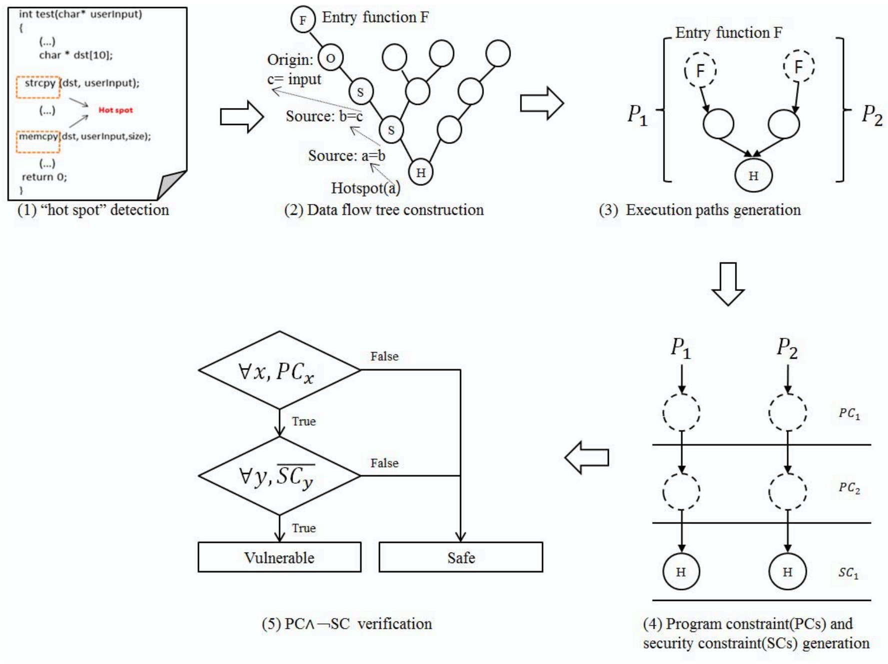
B. Data flow tree construction

In order to eliminate the paths we need to analyze, we trace the sensitive data used in hot spot backwardly to build a data flow tree. Actually, during the process of constructing the tree, paths which are not related to hot spot or sensitive data have automatically been eliminated.

Whats more, we can easily see the relationship between user inputs and function arguments to check whether the arguments in hot spot are tainted by user inputs. The tree is used for finding possible execution paths.

Starting from each hot spot, we trace the arguments until we reach the origin of the sensitive data. In order to understand the process of data flow tree construction, there are several terms that we need to know.

- **Variable Source(S)**: the code statements where we get the source of the variable.



```
int test(char* userInput)
{
      (...)
      char * dst[10];

 strcpy (dst, userInput);

      (...)

memcpy(dst, userInput,size);

      (...)
 return 0;
}
```

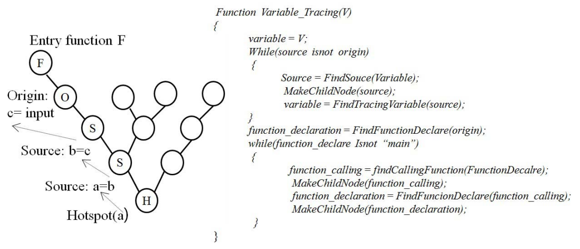
Figure 1. A schematic overview of our mechanism

  - The code statements where the value of the variable is assigned; V_a = expression.
  - Function declaration statements where the variable is used as argument. V_a = void f(int V_a,...,).
- **Variable origin(O)**: the code statement where the value of the variable does not rely on any other variable. Variable origin is also variable source. Variable source could be variable origin.
  - The code statements where the variable is assigned by a constant value; V_a = const.
  - The code statements where the variable gets its value from input files, network messages or command line options; V_a = readfile(); scanf(”s%”,& V_a).
  - The code statements where the variable is initiated; char V_a = ’C’.

We make the hot spot statement as the root of our data flow tree. The critical part of building data flow tree process is tracing variables. This process could be explained as follows:

1) Find the variable need to be traced in a tree node.
2) For the variable need to be traced in tree node A, we find its source and put it into As child node B. If the sources of the variable are in n different exclusive branches, we put the n sources into As n child nodes respectively.
3) Repeat step 1 until the sources in a tree node become origins.
4) After finding the origin, we continue to build the child nodes by function definition-calling logic until the reaching of program entry function such as main

Since this is a very important process during data flow tree construction, we try to make this process more explicitly. The tracing algorithm and process are shown in Figure 2.



```
Function Variable_Tracing(V)
{
        variable = V;
        While(source isnot origin)
        {
                Source = FindSouce(Variable);
                MakeChildNode(source);
                variable = FindTracingVariable(source);
        }
        function_declaration = FindFunctionDeclare(origin);
        while(function_declare Isnot "main")
        {
                function_calling = findCallingFunction(FunctionDecalre);
                MakeChildNode(function_calling);
                function_declaration = FindFuncionDeclare(function_calling);
                MakeChildNode(function_declaration);
        }
}
```

Figure 2. Key process for constructing the data flow tree

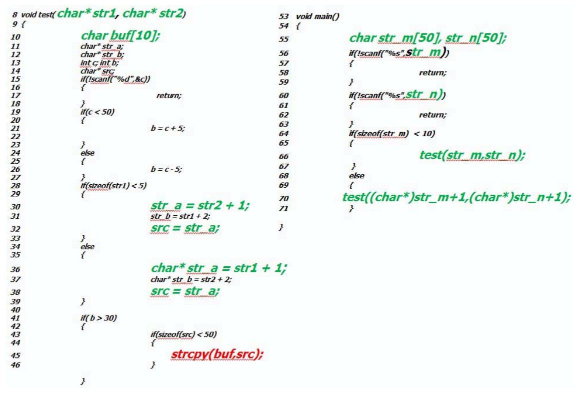
Figure 3 shows the sample code for constructing the data flow tree and Figure 4 shows the data flow tree which comes from the source code sample.

C. The generation of possible execution paths

After the construction of data flow tree, we could simply get possible execution paths from the data flow tree. Figure 5 shows the possible execution paths generated from the previous sample. Take a look at the data flow tree we got from the sample code. Then, we could get 4 main execution paths from this tree. There are:(numbers mean the code lines)

- $53 \to 60 \to 67 \to 8 \to 30 \to 32$
- $53 \to 60 \to 70 \to 8 \to 30 \to 32$
- $53 \to 60 \to 67 \to 8 \to 36 \to 38$
- $53 \to 60 \to 70 \to 8 \to 36 \to 38$

These execution paths are not complete execution paths. They may have more branches. However, these are several main paths. Because for those small branches, most of them are not related to the sensitive data, so we had better not consider them in order to mitigate the paths explosion problem.

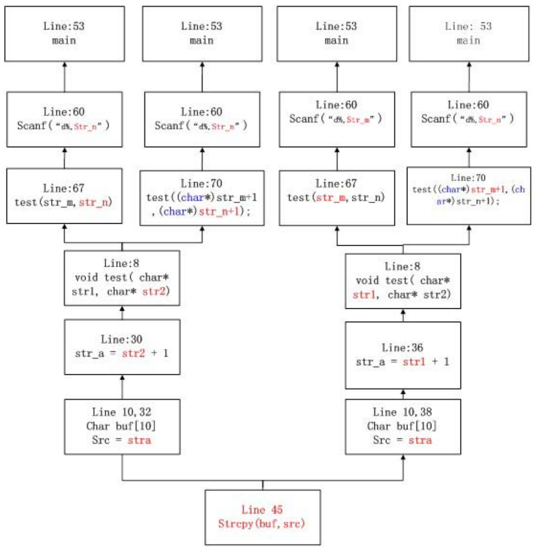


```
8 void test( char* str1, char* str2)
9 {
10              char buf[10];
11              char* str_a;
12              char* str_b;
13              int c; int b;
14              char* src;
15              if(!scanf("%d",&c))
16              {
17                              return;
18              }
19              if(c < 50)
20              {
21                              b = c + 5;
22
23              }
24              else
25              {
26                              b = c - 5;
27              }
28              if(sizeof(str1) < 5)
29              {
30                              str_a = str2 + 1;
31                              str_b = str1 + 2;
32                              src = str_a;
33              }
34              else
35              {
36                              char* str_a = str1 + 1;
37                              char* str_b = str2 + 2;
38                              src = str_a;
39              }
40
41              if( b > 30)
42              {
43                              if(sizeof(src) < 50)
44                              {
45                                  strcpy(buf,src);
46                              }

                }
```

```
53 void main()
54 {
55              char str_m[50], str_n[50];
56              if(!scanf("%s",str_m))
57              {
58                              return;
59              }
60              if(!scanf("%s",str_n))
61              {
62                              return;
63              }
64              if(sizeof(str_m) < 10)
65              {
66                              test(str_m,str_n);
67              }
68              else
69              {
70           test((char*)str_m+1,(char*)str_n+1);
71              }

   }
```

Figure 3. The sample code for constructing the data flow tree

Actually, this is how we solve path explosion problem in our method. Along each possible execution path, we symbolically execute the program to generate program constraints which could be generated so far.



Line:53 main
Line:53 main
Line:53 main
Line: 53 main
Line:60 Scanf("d%,Str_n")
Line:60 Scanf("d%,Str_n")
Line:60 Scanf("d%,Str_m")
Line:60 Scanf("d%,Str_n")
Line:67 test(str_m, str_n)
Line:70 test((char*)str_m+1 , (char*)str_n+1);
Line:67 test(str_m, str_n)
Line:70 test((char*)str_m+1, (char*)str_n+1);
Line:8 void test(char* str1, char* str2)
Line:8 void test(char* str1, char* str2)
Line:30 str_a = str2 + 1
Line:36 str_a = str1 + 1
Line 10,32 Char buf[10] Src = stra
Line 10,38 Char buf[10] Src = stra
Line 45 Strcpy(buf, src)

Figure 4. An example of data flow tree

D. Program constraints (PCs) and security constraints (SCs)

Symbolic execution to generate program constraints (PCs): Symbolic execution is a kind of program analysis method that simulates program execution by replacing concrete values with symbolic variables for inputs.

We use the symbolic executor to symbolically execute the trace to capture program constraints and check the pattern of each executed statement against the security requirements. Especially, when there is a condition statement along the execution paths, we could generate corresponding program constraints according to the current execution path.

Along each possible execution path, we symbolically execute the program to generate program constraints which could be generated so far. We take the 1st path as an example. The following table shows the process of generating the program constraints through path1.

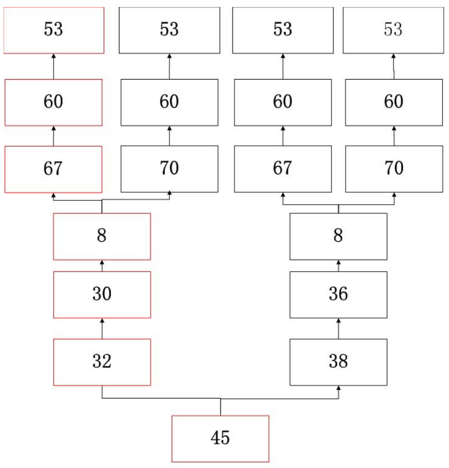


Figure 5. The generation of possible vulnerable paths

- In the case of path: 53 → 60 → 67 → 8 → 30 → 32

Table I
PROGRAM CONSTRAINT GENERATION PROCESS

Path1	**Constraints**
53 → 60	$!scanf("\%s", str_m) = true$ $!scanf("\%s", str_n) = true$
60 → 67	$sizeof(str_m) < 10$
67 → 8	$None$
8 → 30	$!scanf("\%d", \&c) = true$ $sizeof(str1) < 5$
30 → 32	$None$
32 → 45	$b > 30$ $sizeof(src) < 50$

Until now, we get a set of rough program constraints from path1. We call it a rough set because there might be more constraints on this path we haven't found yet. So, we need to find all of the program constraints in order to make our detection result more precise.

- Generate program constraints furthermore.

To explain this part, lets understand what are traced variable and un-traced variable.

Traced variable: variables we have already traced, we know its *origin*.

Un-traced variable: variables we have not traced, we do not know its *origin*.

The set of constraints we got from the previous table is not enough because there might be some un-traced variables in the constraint set. So we will trace the un-traced variables until we find their origins. In the process of tracing un-traced variables, we might get more branches.

This will cause the previous rough path split into more accurate paths. This process will be repeated until there is no un-traced variable in a constraint set. In our example, we can see that variable b in the constraint set is the un-traced variable. So we trace the origin of variable b.

Figure 6 shows the branch caused by tracing the variable b. In path11 and path22, there are already no un-traced variables, so these constraints could be considered as final path constraints. If there is still un-traced variable, then we still have to trace those un-traced variables until there are no un-traced variables in any path constraints.

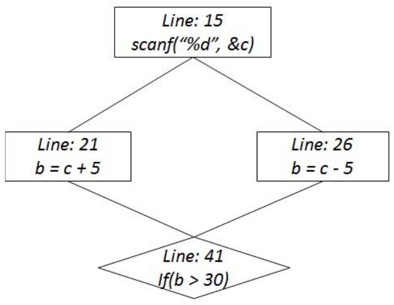


Figure 6. Branches derivated from tracing un-traced variable

From this tree, we know they are 2 more branches. Then we could split Path1 into Path11 and path12. Table II shows the constraints in more detail.

Table II
MORE COMPLETE CONSTRAINT SET

Path11 constraints	**Path12 constraints**
$!scanf("\%s", str_m) = true$	$!scanf("\%s", str_m) = true$
$!scanf("\%s", str_n) = true$	$!scanf("\%s", str_n) = true$
$sizeof(str_m) < 10$	$sizeof(str_m) < 10$
$!scanf(\%d, \&c) = true$	$!scanf(\%d, \&c) = true$
$sizeof(str1) < 5$	$sizeof(str1) < 5$
$sizeof(src) < 50$	$sizeof(src) < 50$
$C < 50$	$C > 50$
$C + 5 > 30$	$C - 5 > 30$

- Security constraints(SCs) generation

Now, we talk about security constraints(SC). Security constraints come from our predefined security requirements. The following table has shown the predefined security requirements for some critical security functions. For each security critical function, we define a corresponding security constraint. In our example:

```
if( b > 30) {
    if(sizeof(src)< 50)
    {
        strcpy(buf, src);
    }
}
```

The security constraint is: $buf.space > src.strlen$

Table III
PREDEFINED SECURITY CONSTRAINTS

Security sensitive functions	**Security requirements**
$strcpy(dst., src)$	$Dst.spcace > src.strlen$
$strncpy(dst, src, n)$	$(dst.space = n) \wedge (n = 0)$
$strcat(dst, src)$	$Dst.space > dst.strlen + src.strlen$
$getcwd(buf, size)$	$(buf.space = size) \wedge (size = 0)$
$fgets(dst, size, f)$	$(dst.space = size) \wedge (size = 0)$
$printf(format, ...)$	$\#offormats = \#ofparameters - 1$

E. PC and SC verification

We use a satisfiability checker to check if $PC \wedge \overline{SC}$ is satisfiable. We have found a vulnerability if the condition could be solved. The solution given by the satisfiability checker is then used to generate test data to uncover the vulnerability. We express both program and security constraints using the SMT-LIB format and use the Yices SMT-solver as the satisfiability checker. Figure 7 shows how the checker works briefly.

IV. IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this section, we will talk about the implementation and our experimental results. We tested the effectiveness of our vulnerability detection approach on sample programs containing buffer overflows and format string bugs which we chose from Juliet Test Suite v1.1 for C/C++ test cases.

Juliet Test Suite is created by US National Security Agency's(NSA) Center for Assured Software. They were made especially for testing the effectiveness of software static analysis tools.

We measured the number of correctly identified vulnerabilities, as well as the number of false positives and false negatives in the results. Ideally, a static analysis tool would be able to find all vulnerabilities in a program without missing any or reporting safe code as vulnerable.

The imprecise nature of static analysis makes this impossible. We can expect a certain number of false positives and false negatives from all vulnerability detection tools. False negatives indicate a failure of the analysis, because a real vulnerability was not detected.

A tool which detects more vulnerabilities is better than a tool reporting false negatives instead. The number of false positives is also important, because they increase the effort required to verify the results of the analysis.

When comparing different vulnerability detection approaches, the number of false positives and false negatives is a good indicator of their effectiveness.

A less important factor in vulnerability detection is the speed of the analysis. However, the speed can be only measured when we test on more complex programs(usually over 1000 code of lines).

We could not measure this aspect since we only did experiments on sample programs. Juliet Test Suite v1.1 for C/C++ test cases shown in figure 7 were created by the US NSAs Center for Assured Software (CAS) specifically for use in testing static analysis methods and tools.

It is intended for anyone who wishes to use the test cases for their own testing purposes. We chose Juliet Test Suite to test our approach over natural code due to the following reasons.

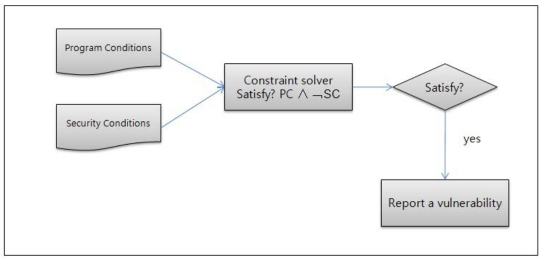
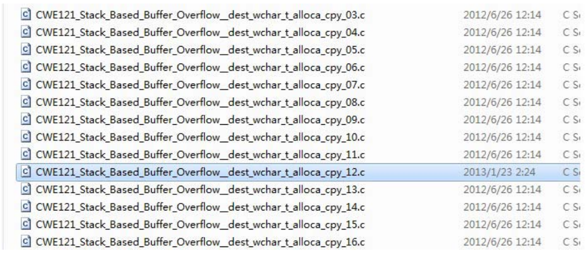


Figure 7. Workflow of a satisfiability checker

1) Evaluating tool results to determine their correctness
When a static analysis tool is run on natural code, each result needs to be reviewed to determine if the code in fact has the specified type of flaw at the specified location (i.e. if the result is correct or a false positive). This review is non-trivial for most results on natural code and often the correctness of a given result cannot be determined with a high degree of certainty in a reasonable amount of time.
2) Comparing results from different tools
Comparing static analysis tool results on natural code is complicated because different tools report results in different manners.
3) Identifying flaws in the code that no tools find
When evaluating static analysis tools, a standard list of all flaws in the code is needed in order to identify which flaws each tool failed to report. On natural code, creating this ”standard” is difficult, especially identifying flaws that are not reported by any automated tool.



```
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_03.c    2012/6/26 12:14    C S
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_04.c    2012/6/26 12:14    C S
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_05.c    2012/6/26 12:14    C S
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_06.c    2012/6/26 12:14    C S
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_07.c    2012/6/26 12:14    C S
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_08.c    2012/6/26 12:14    C S
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_09.c    2012/6/26 12:14    C S
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_10.c    2012/6/26 12:14    C S
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_11.c    2012/6/26 12:14    C S
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_12.c    2013/1/23 2:24     C S
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_13.c    2012/6/26 12:14    C S
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_14.c    2012/6/26 12:14    C S
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_15.c    2012/6/26 12:14    C S
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_16.c    2012/6/26 12:14    C S
```

Figure 8. Test cases from Juliet Test Suites

4) Evaluating tool performance on constructs that do not appear in the code
Natural code has the limitation that even a combination of different projects will likely not contain all flawed and non-flawed constructs that the CAS wants to test.

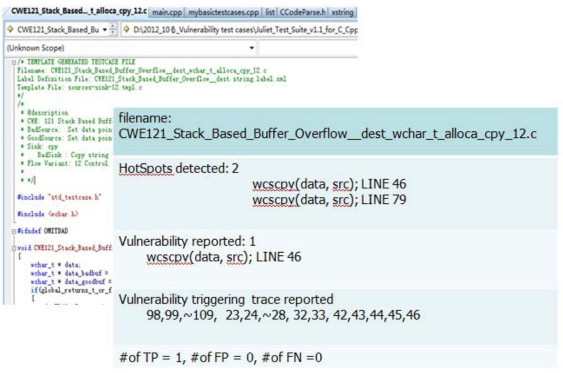
Based on these experiences and challenges, we decided to use Juliet test cases to test and analyze our vulnerability detection approach to control, identify, and locate the vulnerabilities and non-vulnerabilities included in the code.

A. The results of our approach

For testing purposes, we randomly selected 20 test cases out of 854 in the Juliet Test Suite of the buffer overflow type. The results of our sample programs are presented below. We measure the effectiveness of our approach by $Precision$, $Recall$ and $F1\_Value$.

- $Precision$: It is an indicator to measure the preciseness of a certain detection tool. It is calculated by
$$Precision = \frac{TP}{(TP + FP)}. \qquad (1)$$
TP means True Positive and FP means False Positive
- $Recall$: It is an indicator to measure the completeness of a certain detection tool. It is calculated by
$$Recall = \frac{TP}{(TP + FN)}. \qquad (2)$$
TP means True Positive and FN means False Negative.
- $F1\_Value$: It is a comprehensive indicator to measure the combine results of Precision and Recall. It is calculated by
$$F1\_Value = \frac{2 * P * R}{(P + R)}. \qquad (3)$$
P means Precision and R means Recall

Figure 9 shows one of the vulnerability reports generated by our approach and Table 4 shows the testing results of our approach.



CWE121_Stack_Based..._t_alloca_cpy_12.c
main.cpp
mybasictestcases.cpp
list
CCodeParse.h
xstring
CWE121_Stack_Based_Bu
D:\2012_10ǒ_Vulnerability test cases\Juliet_Test_Suite_v1.1_for_C_Cpp
(Unknown Scope)

```
/* TEMPLATE GENERATED TESTCASE FILE
Filename: CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_12.c
Label Definition File: CWE121_Stack_Based_Buffer_Overflow__dest.string.label.xml
Template File: sources-sink-12.tmpl.c
*/
/*
 * @description
 * CWE: 121 Stack Based Buff
 * BadSource:  Set data poin
 * GoodSource: Set data poin
 * Sink: cpy
 *    BadSink : Copy string
 * Flow Variant: 12 Control
 *
 * */

#include "std_testcase.h"

#include <wchar.h>

#ifndef OMITBAD

void CWE121_Stack_Based_Buff
{
    wchar_t * data;
    wchar_t * data_badbuf =
    wchar_t * data_goodbuf =
    if(global_returns_t_or_f
    {
```

filename:
CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_alloca_cpy_12.c

HotSpots detected: 2
wcscpy(data, src); LINE 46
wcscpy(data, src); LINE 79

Vulnerability reported: 1
wcscpy(data, src); LINE 46

Vulnerability triggering trace reported
98,99,~109, 23,24,~28, 32,33, 42,43,44,45,46

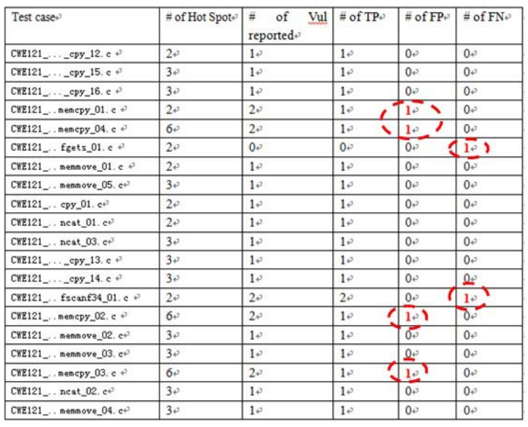
#of TP = 1, #of FP = 0, #of FN =0

Figure 9. Vulnerability report

From figure 10, our approach can get the $Precision$ value of 83.33% , $Recall$ value of 90.90% and $F1\_Value$ of 86.95% which are all relatively high in terms of static analysis tools. We check the 4 False Positives in the results marked by the red circle and we found that all of these 4 test cases have strongly used C-structures in their codes. The FPs are produced due to the inability of our approach to handle struct in C codes. When it comes to the 2 False Negatives we found in this table, these 2 vulnerabilities are not caused by the improper use of security sensitive function which is beyond the detection scope of our approach.

B. Testing competing tools

To evaluate the effectiveness of our system, we tested two competing vulnerability detection tools and compared their results with those of our approach. We chose Flawfinder [14] and SPLINT [15], two programs widely used in the security community.



Test case	# of Hot Spot	# of Vul reported	# of TP	# of FP	# of FN
CWE121_..._cpy_12.c	2	1	1	0	0
CWE121_..._cpy_15.c	3	1	1	0	0
CWE121_..._cpy_16.c	3	1	1	0	0
CWE121_..memcpy_01.c	2	2	1	1	0
CWE121_..memcpy_04.c	6	2	1	1	0
CWE121_.. fgets_01.c	2	0	0	0	1
CWE121_.. memmove_01.c	2	1	1	0	0
CWE121_.. memmove_05.c	3	1	1	0	0
CWE121_.. cpy_01.c	2	1	1	0	0
CWE121_.. ncat_01.c	2	1	1	0	0
CWE121_.. ncat_03.c	3	1	1	0	0
CWE121_..._cpy_13.c	3	1	1	0	0
CWE121_..._cpy_14.c	3	1	1	0	0
CWE121_.. fscanf34_01.c	2	2	2	0	1
CWE121_..memcpy_02.c	6	2	1	1	0
CWE121_.. memmove_02.c	3	1	1	0	0
CWE121_.. memmove_03.c	3	1	1	0	0
CWE121_..memcpy_03.c	6	2	1	1	0
CWE121_.. ncat_02.c	3	1	1	0	0
CWE121_.. memmove_04.c	3	1	1	0	0

Figure 10. Testing results of our approach

Flawfinder is an example of the pattern matching approach to vulnerability detection, while SPLINT uses annotation based data-flow analysis. We performed the tests using the same vulnerable programs from Juliet Test Suite presented earlier in this chapter. We compare our approach, Flawfinder and SPLINT by comparing the number of detected vulnerabilities, false positives and false negatives and $F1\_Value$ reported by each program.

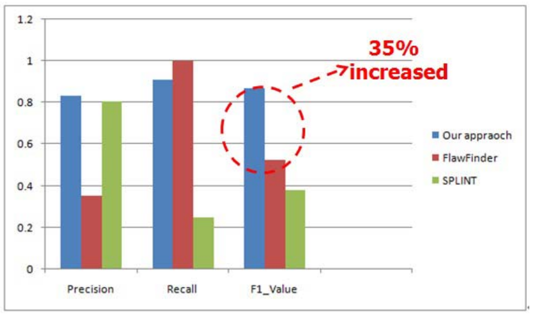


Figure 11. The comparison of 3 approaches testing on the test cases

As we can see from the above figure, when testing on the same group of test cases, our approach has the highest precision compare to the other two. In terms of recall, FlawFinder has the best performance, however, at the cost of being the lowest in precision. When it comes to $F1\_Value$, a more comprehensive indicator, our approach exceeds the FlawFinder by 35% while SPLINT being the lowest. Generally, our approach achieves the best performance among these three tools.

C. Path reduction rate

Since the analyzing time is closely related to the total number of paths we need to analyze, we evaluate the efficiency of our approach by the path reduction ratio. The table 4 gives the definition of parameters that measure the path reduction ratio.

Table IV
PARAMETER EXPLANATION

Parameter	Description
N_α	The total number of bifurcations in the program
N_β	The total number of branches in the program
N_γ	The total number of paths in the program
θ	The percentage of the branches which are related to the hot spot over the total number of branches
ρ	The percentage of the number of eliminated paths over the total number of paths

Figure 12 describes the theoretical path reduction ratio in both best and worst cases. Given that the branches are uniformly distributed in bifurcations, the minimum and maximum of eliminated paths number which we called the worst case and the best case respectively are shown in figure 12.

The result shows that even program paths are exponentially increased, we can efficiently eliminate the number of paths. when the taint ratio is less than 0.4, path reduction ratio ranges from 93.6% in the worst case to 99% in the best case. Normally, taint ratio of most programs is not over 0.4 since taint variables related to hot spot consist only a small portion of the codes in real-world programs.

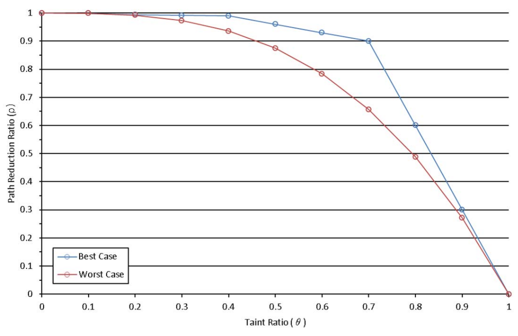


Figure 12. The path reduction ratio by taint variable ratio

In our test cases, we computed the average taint ratio whose value is about 0.38. As a consequence, our method can efficiently mitigate path explosion problem in traditional symbolic execution.

V. CONCLUSION AND LIMITATIONS

In conclusion, we proposed a mechanism to detect software vulnerabilities by using source code analysis and symbolic execution. Different from other static analysis, our approach takes program execution conditions into consideration so that we could dramatically the false positives.

We mitigate the path exploration problem in conventional symbolic execution by considering only vulnerability related paths. Tracing the sensitive data used in hot spot backwardly to recover the execution paths using data flow tree makes us only focus on branches related to sensitive data.

This makes the detection process more efficient. What's more, the $PC \wedge \overline{SC}$ constraint solving makes formal reasoning applicable to accurately detect software vulnerabilities which make us free from bug triggering input generation. To verify our approach, we implemented our mechanism and have it test with source code test cases from Juliet Test Suites.

The results show that our approach achieves relatively high precision and recall. Moreover, practically speaking, our approach can detect vulnerabilities in a reasonable time. However, there are also several aspects we need to consider more.

First, our approach cannot cover some types of vulnerabilities due to the incapability of handling vulnerabilities unrelated to security-sensitive functions.

Then, since we haven't made our program to handle complex data types such as structures in c code, the improvement is needed to make it applicable to complex real world programs.

Finally, since some programs have puzzle functions such as decryption functions and checksum functions which become obstacles when we solve constraints during symbolic execution, we will also need to devote our effort into this in the future.

VI. Acknowledgements

This research was supported by the MKE(The Ministry of Knowledge Economy), Korea and Microsoft under IT/SW Creative research program supervised by the NIPA(National IT Industry Promotion Agency) (H0503-12-1034). Additionally, this research was funded by the MSIP(Ministry of Science, ICT & Future Planning), Korea in the ICT R&D Program 2013.

References

[1] Dazhi Zhang, *Detecting Vulnerabilities in C Programs Using Trace-Based Testing*. Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2010.

[2] D. Wagner, J. Foster, E. Brewer and A. Aiken, *A first step towards automated detection of buffer overrun vulnerabilities*. Proceedings of Network and Distributed System Security Symposium (NDSS), 2000.

[3] C. Cadar and D. Engler, *Execution generated test cases :How to make systems code crash itself*. Proceedings of the International SPIN Workshop on Model Checking of Software, 2005.

[4] K. Sen, D. Marinov and G. Agha, *CUTE: a concolic unit testing engine for C*. Proceedings of the joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005.

[5] E. Haugh and M. Bishop, *Testing C programs for buffer overflow vulnerabilities*. Proceedings of the Network and Distributed System Security Symposium (NDSS), 2003.

[6] R. Hastings and B. Joyce, *Purify: Fast detection of memory leaks and access errors*. Proceedings of the Winter USENIX Conference, 1992.

[7] G. Fink, C. Ko, M. Archer and K. Levitt, *Towards a property-based testing environment with applications to security-critical software*. Proceedings of the 4th Irvine Software Symposium, 1994.

[8] A. Ghosh, T. O'Connor and G. McGraw, *An automated approach for identifying potential vulnerabilities in software*. Proceedings of the IEEE Symposium on Security and Privacy, 1998.

[9] P. Godefroid, N. Klarlund and K. Sen, *DART : directed automated random testing*. Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation, 2005.

[10] M. Ringenburg and D. Grossman, *Preventing format-string attacks via automatic and efficient dynamic checking*. Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2005.

[11] J. Viega, J. T.Bloch, Y. Kohno and G. McGraw, *ITS4: A static vulnerability scanner for C and C++ code*. Proceedings of the Annual Computer Security Applications Conference(ACSAC), 2000.

[12] E. C.Sezer, P. Ning, C. Kil and J. Xu, *Memsherlock: an automated debugger for unknown memory corruption vulnerabilities*. Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2007.

[13] Wheeler and David, *Flawfinder*. http://www.dwheeler.com/flawfinder/

[14] Evans and David, *SPLINT*. http://www.splint.org/,

[15] E. C.Sezer, P. Ning, C. Kil and J. Xu, *Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking*. Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, 2008.

[16] P. Godefroid, M. Y.Levin and D. Molnar, *Automated white-box fuzz testing*. Proceedings of the Network and Distributed Systems Security (NDSS), 2008.

[17] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill and D. Engler, *EXE: Automatically generating inputs of death*. Proceedings of the ACM Conference on Computer and Communications, 2006.

[18] C. Cadar, D. Dunbar and D. Engler, *Automatically generating inputs of death*. Proceedings of the ACM Conference on Computer and Communications, 2008.