Test Case Generation Assisted by Control Dependence Analysis

Puhan Zhang
China Information Technology Security Evaluation Center
Beijing, China
zhangph2008@gmail.com

Qi Wang
Renmin University of China
Beijing, China
China Telecom Corporation Beijing Company
Beijing, China
wangq@163.com

Guowei Dong
China Information Technology Security Evaluation Center
Beijing, China
dgw2008@163.com

Bin Liang, Wenchang Shi
School of Information
Renmin University of China
Beijing, China
{liangb, wenchang}@ruc.edu.cn

***Abstract*—The paper proposes and develops a new test case generation tool named Symbolic Execution & Taint Analysis (SYTA) that can capture implicit information flows by control dependence analysis. When running, SYTA traces execution paths to track constraints on symbolic variables. Some equivalence relationship asserts will be constructed to store the equivalence information among variables for control dependence analysis. If a security sink is reached, SYTA builds a constraint, path conditions and equivalence relationship asserts, which are to be sent to a constraints solver. The test cases will be generated from possible counterexamples in constraint solving. Compared with traditional static analysis tools, SYTA can track implicit information flows, and generate test cases by control dependences analysis effectively.**

Keywords-test case generation; control dependence; implicit information flow; symbolic execution

I. INTRODUCTION

Nowadays, test case generation has become the most important step of code testing, which is usually realized by the symbolic execution approach. If there exists a bug, the test cases can help programmers to find the spot that causes the error.

A traditional Fuzzing approach is a form of blackbox testing which randomly mutates well-formed inputs and use these variants as test cases [1][2]. Although Fuzzing can be remarkably effective, the limitations of Fuzzing are that it usually provides low code coverage and cannot drive deeper into programs because blind modification destroys the structure of inputs [3]. In a security context, these limitations mean that potentially serious security bugs, such as buffer overflows, are possibly missed because the code containing the bugs is even not exercised.

Combining general static analysis with taint analysis to test applications and draw test cases is presently the hottest research technique, such as TaintScope [6]. Taint analysis allows a user to define the taint source and propagate the taint following specific propagation policy during execution, and finally, trigger a particular operation if the predetermined security sink is hit.

Unfortunately, this smart Fuzzing technique bears many pitfalls [4], among which missing the implicit information flows is the most critical one. Contrary to explicit information flows caused by direct assignment, implicit information flows are a kind of information flow consisting of information leakage through control dependence. The example shown in Figure 1 discloses the nature of implicit information flows. There is no direct assignment between variables *h* and *l* in the sample program, but *l* can be set to the value of *h* after the if-then-else block by control dependence. Even though the early attention and definition of the implicit flow problem dated back to 1970's [5], no effective solution has been found. Some newly-developed tools, such as TaintScope [6], detour implicit information flows and limit their analysis only to explicit information flows, which incur the following three problems:

- Missing implicit information flows may lead to a under-tainting problem and false negative. As a result, the security vulnerabilities caused by control dependence will not be detected. Especially, it is critical to capture implicit flows in privacy leak analysis.
- Control dependence is also a common programming form in benign programs. For example, some routines may use a switch structure to convert internal codes to Unicode in a Windows program such as the following code segment. switch(x){ case a: y = a; break; case b: y = b; break; ……}. It indicates that it is necessary to analyze the implicit information flows for common software testing.
- To counter the Anti-Taint-Analysis technique, implicit information flows must be analyzed effectively [7]. Malware can employ control dependence to propagate sensitive information so as to bypass traditional taint analysis.

To address these limitations and generate test cases with tainting techniques, we propose and develop a new tool called Symbolic Execution & Taint Analysis (SYTA), which can generate test cases by considering implicit information flows. Compared with traditional static analysis tools, SYTA can track implicit information flows and generate test cases

```
1:h := h mod 2;
2:if h = 1 then
3:   l := 1;
4:else
5:l := 0;
6:end if
```

Figure 1. A sample program of implicit information flow

by control dependences analysis effectively. Though it is hard to say what percentage of a program can be classified as implicit information flow, it may reveal some vulnerabilities that explicit information flow is unable to.

The rest of the paper is organized as follows. Section 2 briefly analyzes the target problem. Section 3 discusses our methodology and design of SYTA. Section 4 evaluates our approach. Section 5 summarizes related work. Finally, Section 6 concludes the paper and discusses future-work directions.

II. PROBLEM ANALYSIS

This section describes the problem we encounter by walking the readers through the testing of a sample program shown in Figure 3 (a). Despite its small size, it illustrates the most common characteristics of implicit information flows. There exist three bugs related to control dependence in the sample program.

1) Array bound overflow in line 29. The program implies that variable *k* will be equal to variable *i* under a specific condition. If 2 is assigned to *i* by users, *k* will be set to 2 through four control branches, including three 'if' and one loop statements. In line 28, the value to which pointer *p* points is 4. Eventually, an array bound overflow will be triggered when dereferencing *p* as the index of array *a* in line 29.

2) Divide-by-zero in line 30. If 3 is assigned to variable *i* by users, through several control branches, **p* will be set to 0 in line 28, then the divisor *t* becomes 0 in line 30.

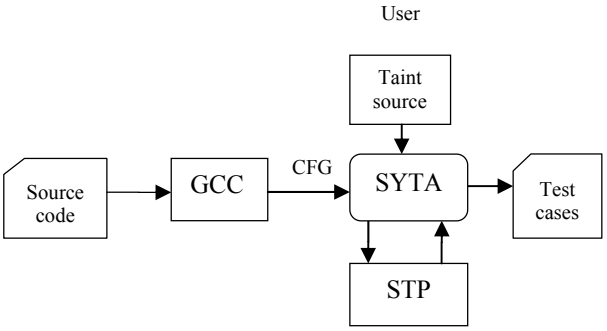
3) Denial of service in line 31. If 1 is assigned to variable *i*, the result of a DoS attack may occur in the program in line 31.

In traditional analysis tools, the test cases cannot be generated for above bugs due to the absence of control dependence analysis. Take EXE [8] and KLEE [9] as examples, they are totally based on explicit information flows analysis. When being applied to the sample program, though variable *i* is symbolically executed and analyzed, those tools can not produce effective test cases, because there are not any direct assignment relationships among *i* and some other variables, such as *k*, *t*, and *p* etc.

Our solution is to take implicit information flows into consideration, in which the flow of taint is propagated from variable *i* to variables *j*, *tmp* and *k* (in line 18, 25 and 30, respectively, in the source code). Variable *p* in line 33 is tainted because of the data flow and it is possible to identify the bugs and automatically obtain the test case to hit them.

III. METHODOLOGY

SYTA, as a test case generator, actually functions as a combination of an intermediate language interpreter, a symbolic execution engine and a taint analyzer. During each symbolic execution, some lists are built to store information for taint propagation. Test programs are firstly parsed by a compiler front-end and converted to an intermediate language. The corresponding Control Flow Graphs (CFGs) are constructed as the inputs of SYTA. SYTA will traverse each CFG and run symbolic execution. It will perform two kinds of taint propagations during symbolic execution, collect symbolic path conditions, record the equivalence information among variables and generate Satisfiability Modulo Theories (SMT) constraints eventually. An SMT solver will be employed to solve and check these constrains to detect potential bugs. If some bugs are found, test cases will be generated and reported.



Note: STP is an SMT solver.

Figure 2. The architecture of SYTA

A. Overview

The core of SYTA is an interpreter loop that selects a path, composed of basic blocks and directed by edges of CFG, to symbolically execute each statement of the basic blocks and perform two kinds of taint propagations (explicit and implicit). The loop continues until no basic blocks remain, and generates test cases if some bugs are hit. The architecture is illustrated in the Figure 2.

For two kinds of taint analysis, we maintain the Explicit Information Flow Taint (EFT) lists and Implicit Information Flow Taint (IFT) lists. Besides, an Equivalence Relationships (ER) list is maintained to record equivalence information among variables in condition statements for control dependence analysis.

At the very beginning of testing, users appoint some interested variables as the taint sources which are recorded into the EFT and IFT lists in proper forms. The two lists involve different taint propagation policies that we design for explicit and implicit information flows respectively.

When a security sink is encountered, SYTA will invoke an SMT solver to carry out a query considering the operation related to current security sink. Current path conditions and expressions drawn from the ER list will act as the context of the query, namely, asserts of solving. By running the query, SYTA checks if any input value exists that may cause a bug. If a bug is detected, the SMT solver will produce a counterexample as a test case to trigger the bug.

B. Implicit Information Flow Taint Propagation

The intuition of taint propagation over implicit information flows can be illustrated using a sample program shown in Figure 4.

In this sample program, a conditional branch statement *br*, namely if ($i >= 4$) in line 5, decides which statements *st* should be executed ($j = 5$ in line 6 or $j = 0$ in line 9). The

value of *i* affect the value of *j*. Therefore, based on control dependence, the taint state should be propagated from the source operand of *br*, namely the variable *i*, to *st*'s destination operands, the variable *j*. To achieve this result, SYTA needs to compute and record post dominance relationships at the basic-block level before symbolic execution.

At first, a user appoints variables as taint sources and SYTA calculates the immediate post-dominant basic block of the corresponding basic block containing the taint sources. Insert the pair <*i*, *ipdom_bb*> into the IFT list, where *i* stands for the tainted variable, and *ipdom_bb* means the immediate post-dominate basic block of the current basic block.

During path travelling, when a basic block is reached, SYTA compares it with all the *ipdom_bb*s in the control-flow based taint pairs in the IFT list in an attempt to find matches and then remove the matching pairs. After removing, if the IFT list is not empty, the *ipdom* of the current basic block will be calculated and the taint pairs are formed together with every variable *v* referenced in the current basic block. These pairs are added to the IFT list one by one. In other words, if the target variable *i* is marked as tainted, the variables in the current basic block will also be marked as tainted according to the control dependence relationship. No further operations will be performed if the IFT list is NULL and only the explicit information flow taint propagation goes on.

In a CFG, basic block *m* post-dominates (*ipdom*) *n* means all directed paths from *m* to the exit basic block contain *n*. If there is no node *o* such that *n pdom o* and *o pdom m*, we call *m* is immediate post-dominates *n*. Just like that in Figure 4, BB5*ipdom* BB2.

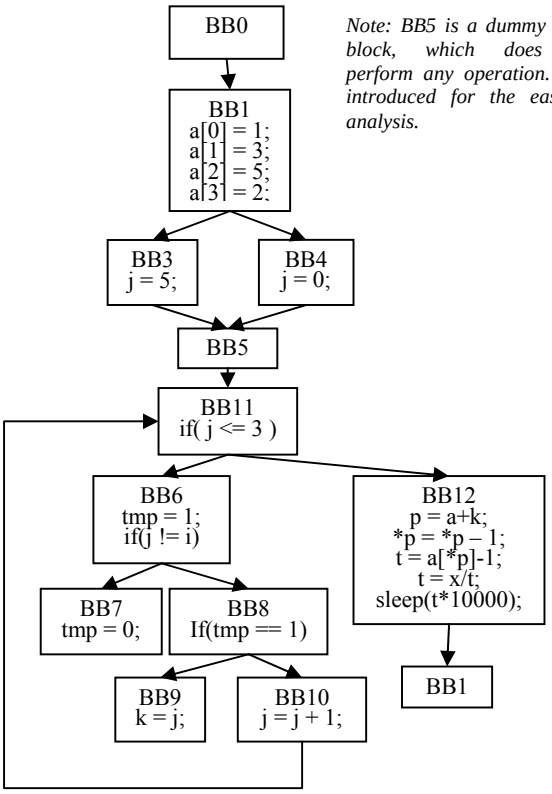
Take the program in Figure 4 (a) as an example again, whose CFG and post-dominance tree are shown in Figure 4 (b) and (c), respectively. We assume that variable *i* is chosen as a taint source. At first, the IFT list is initialized to be empty. When line 5 is executed, SYTA will identify the current statement as a condition statement. The corresponding *ipdom* is BB5, a pair <*i*, BB5> will be added into the IFT list. The symbolic execution forks here and finds both paths are feasible. The true branch would be executed first and line 6 is reached. At this time, the index of the current basic block is 3, and there are not matching pairs in the IFT list. The destination operand of the statement, the left-value *j*, would be added into IFT list together with its *ipdom* BB5 in the form of <*j*, BB5>. All these two pairs will be removed when line 11 is reached because BB5 matches either of them.

C. Explicit Information Flow Taint Propagation

Explicit information flow taint propagation is quite straightforward compared with the implicit one. Only direct data dependence, such as assignment operations, needs to be

```
1:  void main(void) {
2:     unsigned int i;
3:     unsigned int t;
4:     int a[4] = { 1, 3, 5, 1 };
5:     int *p;
6:     int tmp;
7:     int j;
8:     int k;
9:     int x = 100;
10:    scanf("%d",&i);
11:    if(i >= 4){
12:           j = 5;
13:    }
14:    else{
15:           j = 0;
16:    }
17:    for(j; j<4;j++)
18:    {
19:           tmp = 1;
20:           if( j != i){
21:                  tmp = 0;
22:           }
23:           if (tmp == 1){
24:                  k = j;
25:           }
26:    }
27:    p = a+k;
28:    *p = *p - 1;
29:    t = a[*p]-1;
30:    t = x / t;
31     sleep (t*10000);
32:  }
```

(a)



(b)

Figure 3. The source code and CFG of a sample program under testing

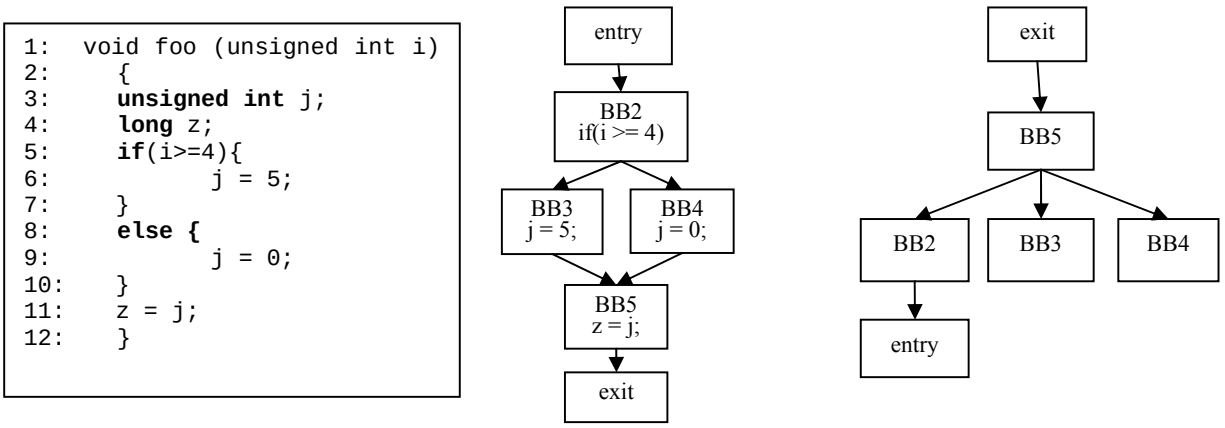


Figure 4. A fragment of the sample program in Figure 3 and its CFG, post-dominate tree

considered in taint propagation.

When an assignment statement is encountered, SYTA will check whether the operands on the right side of the statement are included in the EFT list. If the answer is positive, SYTA will insert the left operand into the EFT list.

In addition, when new pairs are added into the IFT list, the corresponding variables of the pairs should meanwhile be inserted into the EFT list too. This approach is adopted because the information flow among variables maybe proceeds alternately between the two forms. This is a generally ignored problem. Let's still take the case in Figure 4 as an example, in the program, there is no explicit information flow from variable *i*, exists only an implicit information flow from variable *i* to variable *j* caused by the if-then-else clause. But the information flow from *j* to variable *z* in line 11 is explicit. If the two kinds of information flows are processed separately, then in line 11, the variable *z* will not be tainted because the variable *j* is only tainted with implicit information flow. As a result, no taint markings will the variable *z* has, which leads to false negatives because the value of variable *z* is influenced by variable *i*.

D. Test Case Generation

In KLEE, the context of constraints solving only contains path conditions. In order to capture the implicit information flows, the indirect equivalence relationships between variables are also identified by SYTA and sent to the SMT solver as asserts. Take the program in Figure 3 as an example, there exists an implicit equivalence relationship between *k* and *j* (i.e., $k == j$) after executing line 24. When the branch condition is not satisfied in line 20, both the relationships $j == i$ and $k == j$ will hold after line 24. SYTA will record both equivalent variables pairs in the ER list rather than only one explicit pair (i.e., $j == i$).

When a security sink is encountered, two kinds of asserts will be sent to the SMT solver as the context. One is the path condition of the current path, the other is a Conjunctive Normal Form (CNF) formed with pairs in the ER list. As illustrated in the CFG in Figure 3(b), which is expressed in the intermediate language, when the execution path is (BB0 → BB1 → BB4 → BB5 → BB11 → BB6 → BB8 → BB9 → BB10 → BB11 → BB12), the current security sink is a reference to array *a*. At this time, the path condition is ($i \leqslant 3$ && $j \leqslant 3$ && $j == i$ && $tmp == 1$ && $j(1) \geqslant 3$); the assert drawn from the ER list is ($k == i$), *j(1)* is an alias of variable *j*. All these expressions are set to be asserts of the SMT solver. The query submitted is ($*p \geqslant 0$ && $*p \leqslant 3$). The counterexamples the SMT solver provides are ($i = 2$; $j = 2$; $*p = 4$). The test case is ($i = 2$).

Three kinds of bugs are considered in SYTA: (1) array bound overflow, (2) divide-by-zero, and (3) denial-of-service.

(1) If the index variable is marked as tainted in a reference to an array, a query is constructed as (*index* >= 0 && *index* <= *upperbound* – 1) and be sent to the SMT solver. Under certain contexts, there exists an array bound overflow if all the constraints are satisfied and the query is not.

(2) If the operator is a divisor, and the divisor *m* is tainted. Then the sink query (m != 0) is constructed and sent to the SMT solver together with all the asserts gathered till now. Divide-by-zero is found if all the constraints are satisfied and the query is not.

(3) When the function *sleep* is called, and its parameter is marked as tainted, then the query (*sleep* <= 10000) is constructed and sent to the SMT solver together with all the asserts. Then the DoS bug exists if all the constraints are satisfied and the query is not.

In a word, when SYTA encounters a security sink, it will gather all the path conditions preceding the current statement and asserts from ER list, the query will be sent to the SMT solver. If the query is unsatisfied, a test case is generated and reported with the bug name.

Based on the above discussion, as shown in Table I, three test cases are generated to detect bugs in the sample program in Figure 3. They can be used to trigger the array bound overflow, divide-by-zero and DoS bugs, respectively.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-376-6

TABLE I. TEST CASES OF THE SAMPLE PROGRAM BY SYTA

Taint Sources	Tainted Variables	Vulnerability Type
$i = 2$;	$j = 2$; $*p = 4$;	array bound overflow
$i = 3$;	$j = 3$; $*p = 1$; $t = 0$;	divide-by-zero
$i = 1$;	$j = 1$; $*p = 2$; $t = 25$;	dinal-of-service

E. Implementation

As shown in Figure 2, we employ GCC 4.5.0 as the compiler front-end of SYTA. Source code will be parsed and convert to GIMPLE intermediate representation; its CFGs are also built by leveraging GCC. SYTA is implemented as a pass of GCC, analysis will be performed at the GIMPLE level. Finally, we choose the commonly used constraints solver STP [16] as the SMT solver in SYTA.

IV. EVALUATION

We illustrate two cases that show how SYTA can detect errors. In the program shown in Figure 5, the control dependence relationships are based on the *switch-case* structure. During analysis, we leverage the GIMPLE intermediate representation of GCC to process the *switch-case* structure. In GIMPLE, a *switch-case* will be regarded as a normal *if-else* structure. When the original taint source is variable *n*, a counterexample ($n = 245$) can be got and the

```
void foo(int n)

{
  Unsigned int y[256];
  Unsigned int x[256];

  for(int i=0; i<256; i++)
  {
    y[i] = (char)i;
  }

  for(int j=0; j<n; j++)
  {
    switch(y[j])
    {
      case 0:
             x[j] = 13;
             break;
      case 1:
             x[j] = 14;
             break;
      case 2:
             x[j] = 15;
             break;
      ……
      case 256:
             x[j] = 12;
             break;
    }
  }
  n = y[n]/x[n-1];
}
```

Figure 5. The first case study

```
void foo(int h)
{
     int a[5] = {1,2,3,4,5};
     int l = 10;
     int k = 0;
     if(h < 0){
             l = 0;
     }
     while(l != 0){
             if(l <= 5){
                     k++;
             }
             l--;
     }
     l = a[k];
}
```

Figure 6. The second case study

assignment statement ($n = y[n] / x[n\text{-}1];$) may trigger a *divide-by-zero* bug.

In the program shown in Figure 6, there is no explicit *else* branch in the if ($h < 0$) statement. In order to capture the taint propagation through the missing *else* branch, an assisting *else* branch is inserted into the intermediate representation, which includes a dummy statement ($l = l$). Using the dummy statement, a counterexample ($h = 2$) would be found as the test case for the array bound overflow bug at the last statement.

In this paper, we try to extend the test case generation technique to cover implicit information flows rather than only explicit information flows. In theory, it is impossible to track and analyze all forms of implicit information flows. Our study shows that some typical forms of implicit information flows can be effectively tracked to support test case generation. In this section, we employ two proof-of-principle samples to demonstrate the ability of SYTA to track typical forms of implicit information flows. We also use KLEE (with LLVM v2.7) to analyze the two samples and the program shown in Figure 3(a). Compared with SYTA, KLEE, as shown in Figure 7, only provides two test cases (i.e., $i = 1$ and $i = 2147483648$) for feasible execution paths of the program shown in Figure 3(a), but these cases can not trigger and report the array-bound-overflow and divide-by-zero vulnerabilities. Nevertheless, frankly

```
KLEE: output directory = "klee-out-0"
KLEE: done: total instructions = 68
KLEE: done: completed paths = 2
KLEE: done: generated tests = 2
$ ktest-tool --write-ints klee-last/test000001.ktest
  ktest file : 'klee-last/test000001.ktest'
  args       : ['test.o']
  num objects: 1
  object    0: name: 'i'
  object    0: size: 4
  object    0: data: 1

  $ ktest-tool --write-ints klee-last/test000002.ktest
  ...
  object    0: data: 2147483648
```

Figure 7. The KLEE analysis result for the program in Figure 3(a)

speaking, analyzing a large scale real-work system will require much more computing overhead.

V. RELATED WORK

Even though early attention and definition of implicit information flow dated back to 1970's, no effective solution has been found. Lots of newly-developed tools, like TaintScope [6], detour the implicit information flow problem and limit their applications only to explicit information flows. Some other work limits the processing of control dependence to predetermined forms, for example, Heng Yin et al. deal with the API function containing control dependence specially in their tool Panorama [11]; The system designed by Wei Xu et al. [12] process only two specific kinds of control flow; Dongseok Jang et al. [13] only process the branching but not the whole program leading to low coverage and false negatives.

Some dynamic analysis testing tools are more comprehensive, like Dytan [10] by James Clause, it can construct implicit information flow on the binary code but cannot get the control dependence information from indirect jump instructions. In DTA++ developed by Min Gyung Kang et al., information preserving implicit information flows are traced [14], but the simple dichotomy approach is too rough and may cause under-tainting problem.

VI. CONCLUSION AND FUTURE WORK

We presented a static analysis tool, SYTA, capable of automatically generating test cases using symbolic execution and taint analysis techniques. Using the control flow graph of the target program and user-appointed taint sources as inputs, SYTA follows execution paths to track the constraints on symbolic variables, and maintains two taints lists for explicit and implicit information flows respectively. The test cases will be generated from possible counterexamples in a constraint solving process. Compared with traditional static analysis tools, SYTA can track implicit information flows, generates test cases by control dependence analysis effectively.

At present, in tracking implicit information flows, SYTA cover only three kinds of sink points, concerning array bound overflow, divide-by-zero, and denial-of-service, respectively. By expending taint source points and sink points, it may cover other kinds of vulnerabilities related to taint data. For example, by regarding untrusted input interface functions as taint source points and function *memcpy* and the like as sink points, it can detect buffer overflow vulnerability led to by ineffective input validation.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. The work has been supported in part by the National Natural Science Foundation of China (61070192, 61170240, 61272493, 61100047), the Natural Science Foundation of Beijing (4122041), the National Science and Technology Major Project of China (2012ZX01039-004), and the National High Technology Research and Development Program of China (2012AA012903).

REFERENCES

[1] D. Bird and C. Munoz, "Automatic Generation of Random Self-Checking Test Cases," IBM Systems Journal, Vol. 22, No. 3, 1983, pp. 229-245.

[2] Protos, Web page: http://www.ee.oulu.fi/research/ouspg/protos/, [retrieved: August, 2014].

[3] J. Offutt and J. Hayes, "A Semantic Model of Program Faults," in Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis), San Diego, January 1996, pp. 195-200.

[4] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in Proceedings of the IEEE Symposium on Security and Privacy, May 2010, pp. 317-331.

[5] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," Comm. of the ACM, vol. 20, no. 7, July 1977, pp. 504-513.

[6] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in Proceedings of the 31st IEEE Symposium on Security and Privacy, Oakland, California, USA, May 2010, pp. 497-512.

[7] L. Cavallaro, P. Saxena, and R. Sekar, Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. Technical report, Stony Brook University, 2007.

[8] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in Proceedings of the USENIX Symposium on Operating System Design and Implementation, 2008, pp. 209-224.

[9] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: A system for automatically generating inputs of death using symbolic execution," in Proceedings of the ACM Conference on Computer and Communications Security, October 2006, pp.322-335.

[10] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in International Symposium on Software Testing and Analysis, 2007, pp. 196-206.

[11] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in Proceedings of the ACM Conference on Computer and Communications Security, October 2007, pp. 116-127.

[12] W. Xu, E. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in Proceedings of the USENIX Security Symposium, 2006, pp. 121-136.

[13] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in JavaScript web applications," in Proceedings of the ACM Conference on Computer and Communications Security, 2010, pp. 270-283.

[14] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in International Symposium on Software Testing and Analysis, 2007, pp. 196-206.

[15] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation," in Proceedings of the Network and Distributed System Security Symposium, February 2011, pp. 205-219.

[16] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in Proceedings of the 19th International Conference on Computer Aided Verification, 2007, pp. 519-531.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-376-6