Information and Software Technology 80 (2016) 231–244

Contents lists available at ScienceDirect

Information and Software Technology

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

Towards designing an extendable vulnerability detection method for executable codes☆

Maryam Mouzarani, Babak Sadeghiyan*

Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran

ARTICLE INFO

Article history:
Received 16 September 2015
Revised 31 July 2016
Accepted 12 September 2016
Available online 13 September 2016

Keywords:
Software vulnerability
Executable codes
General specification
Extendable method

ABSTRACT

Context: Software vulnerabilities allow the attackers to harm the computer systems. Timely detection and removal of vulnerabilities help to improve the security of computer systems and avoid the losses from exploiting the vulnerabilities.

Objective: Various methods have been proposed to detect the vulnerabilities in the past decades. However, most of these methods are suggested for detecting one or a limited number of vulnerability classes and require fundamental changes to be able to detect other vulnerabilities.

In this paper, we present a first step towards designing an extendable vulnerability detection method that is independent from the characteristics of specific vulnerabilities.

Method: To do so, we first propose a general model for specifying software vulnerabilities. Based on this model, a general specification method and an extendable algorithm is then presented for detecting the specified vulnerabilities in executable codes.

As the first step, single-instruction vulnerabilities–the vulnerabilities that appear in one instruction–are specified and detected. We present a formal definition for single-instruction vulnerabilities. In our method, detection of the specified vulnerabilities is considered as solving a satisfaction problem. The suggested method is implemented as a plug-in for Valgrind binary instrumentation framework and the vulnerabilities are specified by the use of Valgrind intermediate language, called Vex.

Results: Three classes of single-instruction vulnerabilities are specified in this paper, i. e. division by zero, integer bugs and NULL pointer dereference. The experiments demonstrate that the presented specification for these vulnerabilities are accurate and the implemented method can detect all the specified vulnerabilities.

Conclusion: As we employ a general model for specifying the vulnerabilities and the structure of our vulnerability detection method does not depend on a specific vulnerability, our method can be extended to detect other specified vulnerabilities.

1. Introduction

Detecting software vulnerabilities has been studied widely in the past decades. As a result various methods are presented to detect vulnerabilities more accurately, with less false positives and false negatives. However, most of these methods are suggested for detecting one or a limited number of vulnerabilities. Thus, their algorithms should be changed to detect new vulnerabilities [1]. Therefore, the enhanced techniques applied in one algorithm are not usable for detecting other vulnerabilities.

As an example, many advances have occurred in detecting the buffer overflow vulnerability during the past years. Different techniques have been suggested to detect this vulnerability, such as pattern matching [2], program annotation [3,4], constraint analysis [5] and taint analysis [6,7]. If we call the algorithm that analyzes the program and searches for a vulnerability in it as vulnerability seeking algorithm, most of vulnerability seeking algorithms in these works are designed based on the mechanism of the buffer overflow vulnerability. For example, the vulnerability seeking algorithm presented in [5] considers the strings in a C program as an abstract data type with pre-defined functions, such as *str-*

☆ This work is supported in part by APA Research Center of Amirkabir University of Technology (Tehran Polytechnic).

* Corresponding author.
E-mail address: basadegh@aut.ac.ir (B. Sadeghiyan).

http://dx.doi.org/10.1016/j.infsof.2016.09.004
0950-5849/

cpy(), strcat(), etc. The state of each string is also summarized with two integer values, i. e. its allocated size and its current length. Thus, the content of the strings is not important for this algorithm. For each string buffer, the algorithm examines string manipulation statements to check whether the maximum length of a string exceeds its allocated size. If such condition is detected, a buffer overflow vulnerability would be reported. This algorithm requires fundamental changes to be able to detect another vulnerability, such as format string, command injection or dangling pointers.

It is worth mentioning how we differentiate a vulnerability detection technique from a vulnerability seeking algorithm. A vulnerability detection technique is the general instruction for finding vulnerabilities in the programs and is not usually specific to a particular vulnerability. For example, taint analysis is a detection technique that has been used for detecting various vulnerabilities, e. g. SQL injection [8,9], XSS [8,10], buffer overflow [6,7] and format string [11]. A vulnerability seeking algorithm is an accurately defined instruction for analyzing particular programs and seeking specific vulnerabilities based on one or a combination of vulnerability detection techniques. For example, the vulnerability seeking algorithm in [5] is designed based on the constraint analysis technique to detect buffer overflow in C programs. The design of most of the vulnerability seeking algorithms depends on the mechanism of the intended vulnerabilities. For example, a successful buffer overflow seeking algorithm may not be easily used to detect other types of vulnerabilities.

We believe that an extendable vulnerability seeking algorithm could be a solution for this limitation. By an "extendable vulnerability seeking algorithm", we mean an algorithm that is able to detect the specified vulnerability classes in the target program, even the vulnerabilities that are specified in the future. Vulnerabilities have some common characteristics and can be defined in a general structure. Also, there are vulnerability detection techniques that have been used separately for detecting various vulnerabilities, such as taint analysis or symbolic execution [12–14]. Thus, an extendable vulnerability seeking algorithm that is designed based on such techniques may be able to detect different vulnerabilities at the same time.

To be extendable, the vulnerability seeking algorithm should be independent from the specification of the vulnerabilities as much as possible. In this way, the vulnerability seeking algorithm can be improved separately and get benefit from the enhancements in other detection techniques. In this paper, we present a first step towards designing a general extendable vulnerability detection method. This method consists of a general specification model for specifying vulnerabilities in a way that is understandable by the vulnerability seeking algorithm. It also contains an extendable vulnerability seeking algorithm that searches for any specified vulnerabilities in the program automatically.

There are a limited number of extendable vulnerability seeking algorithms and vulnerability specification methods presented by now, that will be reviewed in Section 2. However, there is no extendable vulnerability detection method for executable codes. The advantages of analyzing executable codes, instead of source codes, in detecting software vulnerability have propelled us to take steps in designing an extendable vulnerability detection method for executable codes. Reflection of the exact behavior of the program, optimizations and bugs in the compilers, unavailability of the source codes and platform-specific details are some reasons that make analyzing executable codes more preferable [15]. Moreover, analyzing the executable codes for detecting the vulnerabilities makes the method independent from the development language and thus the method would cover more programs.

In this paper, we consider the vulnerabilities that appear in a single instruction. Therefore, the paper is regarded as a first step towards designing an extendable detection method. Single-instruction vulnerabilities can be specified based on the arguments of one instruction, such as division by zero [16] and some integer bugs [17–19]. Other vulnerabilities that appear in a scenario, in more than one instruction, are not considered in this paper and will be studied in our future works.

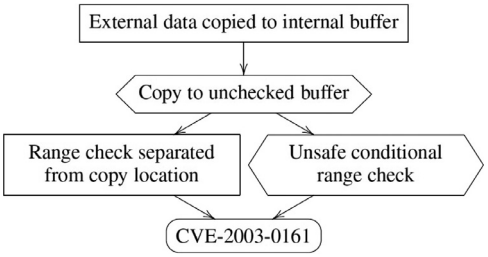

Fig. 1. A sample VCG for CVE-2003-0161 [22].

We present a general model for specifying the vulnerabilities. Based on this model, vulnerabilities are specified by the use of Vex language. Vex is an intermediate representation for the executable codes that is used in Valgrind binary instrumentation framework [20]. The vulnerability seeking algorithm is implemented as a plug-in for Valgrind. It automatically searches in executable codes for any specified single-instruction vulnerability. The ease of extending the method to other single-instruction vulnerabilities will be demonstrated.

Hence, the followings can be considered as the contributions of this paper:

- Presenting a general model for specifying software vulnerabilities.
- Presenting a method for specifying the single-instruction vulnerabilities to be detected in the executable codes.
- Presenting an extendable vulnerability detection method for executable codes based on the proposed specification model.
- Specification and detection of vulnerability classes division by zero [16], NULL pointer dereference [21], integer overflow [17], integer underflow [18] and incorrect width conversion in numeric type (for integer type) [19] using the proposed extendable detection method.

This paper is organized as following: Section 2 reviews the related works. The general model for specifying software vulnerabilities is presented in Section 3. Based on this model, a general extendable vulnerability seeking algorithm is presented in Section 4. Section 5 presents the details of designing and implementing an extendable vulnerability detection method for executable codes. The implemented method is evaluated in Section 6. We conclude the paper and suggest some future works in Section 7.

2. Related works

One of the well-known vulnerability specification methods is called Vulnerability Cause Graph (VCG). A VCG is a directed non-cyclic graph that illustrates how and why a vulnerability appears in a program [22]. It has one leaf node that defines a specific vulnerability. The other nodes are *causes* that explain the conditions and events during the development process that make software vulnerable. Fig. 1 illustrates an example VCG. This graph explains how the vulnerability CVE-2003-0161 is created in Sendmail mail server. Although these graphs help the developers learn about different vulnerabilities, the narrative specification of causes prevents them to be automatically understandable. Thus, this specification method is not usable in an extendable vulnerability detection method.

Mallouli et al. specify vulnerabilities formally in [23] based on Vulnerability Detection Conditions (VDCs). A VDC characterizes

vulnerabilities with three elements: pre-conditions, actions and post-conditions. In fact, a VDC determines that under the specified pre-conditions, if a specific action results in the specified post-conditions, there is a vulnerability in the program. The vulnerability seeking algorithm in this method searches for specified vulnerabilities by monitoring the program execution. For each instruction, the current state, the instruction and the resulted state after execution of the instruction are compared with the specified VDCs. If there is a match, a vulnerability would be reported. This method is designed for detecting specific vulnerabilities in programs written in C, such as the ones reported in CVE database [24]. There are a huge number of specific vulnerabilities recorded in the CVE database. Thus, the vulnerability seeking algorithm has to compare each operation in the program with many VDCs. Since the vulnerability seeking algorithm utilizes a monitoring technique, it imposes a high overhead on the execution of the program. We specify the vulnerability classes, such as the ones defined in CWE [25], instead of the specific ones to make the vulnerability seeking algorithm more efficient. Also, our method is proposed for detecting the vulnerabilities in the executable codes. Thus, it is not limited to a specific programming language and covers a wide range of programs.

A language, called PQL, is applied in [26] to specify vulnerabilities in web-based programs written in Java. It specifies vulnerabilities as a trace of events for the defined objects in a program. The vulnerability seeking algorithm searches statically and dynamically in the program for specified vulnerabilities. To be simple, the language does not support some data types, e. g. integer, float and character. Therefore, certain operations, such as mathematical operations or comparison of the characters, cannot be expressed in the vulnerability specification. This is not a limitation for specifying vulnerabilities in object-oriented programs. Because they encapsulate these operations in certain methods for each data type. It is, however, a limitation for specifying vulnerabilities in other languages like C. For example, it is not possible to specify the integer overflow vulnerability in C language by the use of PQL.

Software vulnerabilities are specified in [1] with the help of a descriptive language called OCL. OCL is a descriptive language based on first-order logic and set theory [27]. In [1] an object-oriented program, including the program components, inputs, outputs, classes, loops, conditional statements, etc. is modeled with OCL. Each object in the model has specific attributes, such as name, type and accessibility. Vulnerabilities are specified based on the defined objects and their attributes. The vulnerability seeking algorithm parses the code statically and extracts its Abstract Syntax Tree (AST). The AST is then translated into an intermediate representation to be comparable with vulnerability specifications. This method is implemented for analyzing programs written in C, C#, C++ and VB.NET.

Our proposed specification method specifies the vulnerabilities formally so the detection algorithm parses them and learns how to detect them in the executable codes. We also present a vulnerability seeking algorithm that detects automatically any specified vulnerabilities in the executable codes. The suggested algorithm does not use a monitoring technique, thus it does not impose an overhead on the program when it is executed by the end user.

Since the proposed method is for specifying the vulnerabilities in executable codes, it can be used to detect vulnerabilities in the programs written by different programming languages. Also, the specification method is not limited to specific data types. Thus, it covers a wide range of software vulnerabilities.

3. A general model

We propose a general model for specifying software vulnerabilities based on our understanding of how a program becomes vulnerable. In an abstract level, the program consists of instructions and data. The instructions perform pre-defined operations on different data. In fact, various behavior of an instruction is the result of manipulating different data. The employed data in that instruction can be controlled by other instructions. For example, one may check the value of data before it is used as the divisor of a division instruction. If the value of data is equal to zero, it would not be used in the division instruction. When a program allows executing the instructions on *inappropriate* data, it means that there is a fault in the program. If the faults allow compromising the security policy, the program is vulnerable.

For each vulnerability, specific data are concerned. For example, for stack overflow vulnerability the return address, local variables, function arguments and the previous stack pointer are considered. If these data change inappropriately, the program becomes vulnerable. The *inappropriate* change is defined based on the definition of the vulnerability class. Hence, we model vulnerabilities with a two-element structure of {Container(s), Rule}. Containers are the data holding entities which are related to the vulnerability. A container may be a variable, a register, a memory area or even a flag. The rule defines inappropriate data assignment to the containers that makes the program vulnerable.

At the first step in designing an extendable detection method, we consider vulnerabilities that appear in a single instruction, such as division by zero, NULL pointer dereference [21] and some integer bugs. These vulnerabilities can be specified based on one instruction, and their containers and the rule are defined according to the elements of that instruction. For example, the division by zero vulnerability may exist in a division instruction, or an integer overflow may occur in an addition instruction. Other vulnerabilities that appear in a scenario, with more than one instruction, will be considered in our future works.

We define a single-instruction vulnerability Vul_x formally as: $Vul_x = \{CONT, f(CONT)\}$. In this definition, $CONT$ is a set of variables in first-order logic and $f(CONT)$ is a first-order logic formula on $CONT$. The formula expresses predicates on the containers that can be true or false. A single-instruction vulnerability appears in an instruction when the data arguments of that instruction hold inappropriate data values. Thus, the vulnerability can be specified by defining the relevant data arguments as the containers, and holding of inappropriate values as a first-order logic formula. In this way, every single-instruction vulnerability can be represented in this general model.

With this definition, a program contains Vul_x if $\exists CONT, f(CONT) = True$. Therefore, detecting a vulnerability Vul_x in a program means finding appropriate data values for the containers that make $f(CONT) = True$. This is equivalent to determining the satisfiability of $f(CONT)$. In fact, $f(CONT)$ is satisfiable if $\exists CONT, f(CONT) = True$. Thus, detecting a vulnerability in a program can be considered as solving a satisfaction problem. In this way, detecting a vulnerability in a program means finding appropriate data values for the containers in that program that satisfy the rule of the vulnerability.

Table 1, shows the specification of these vulnerabilities in our two-element model. The formula column presents our formal representation of the rules for each vulnerability. For the vulnerabilities in the last three rows, a number of possible formulas are presented in this table because of lack of space. The complete set of rules for these vulnerabilities are presented and described in Section 4.1.2.

To detect the specified vulnerabilities, the detection algorithm locates the containers in the target program. For each located container, it checks the satisfiability of the vulnerability rule. If the detection algorithm finds possible values for the containers that satisfy the vulnerability rule, there might be a vulnerability in the program.

Table 1
Informal specification of sample vulnerabilities in the two-element model.

Vulnerability	Description	Container(s)	Rule	Formula
Division by zero	The product divides a value by zero [16].	The divisor argument of a division operator.	The container is equal to zero.	{x=0}
NULL pointer dereference	The application dereferences a pointer that it expects to be valid, but is NULL [21].	The pointers.	The container is equal to NULL.	{x=NULL}
Integer overflow	An integer value is incremented to a value that is too large to store in the associated representation [17].	Integer arguments of arithmetic operations, such as unsigned Add32, Mul32 or signed Add32.	The arithmetic operation on the containers results in value larger than the associated representation.	{x+y>0×00000000ffffffff} (for containers of unsigned Add32) {x× y>0×00000000ffffffff} (for containers of unsigned Mul32) {x<0 AND y<0 AND x+y>0} (for containers of signed Add32)
Integer underflow	The product subtracts one value from another, such that the result is less than the minimum allowable integer value, which produces a value that is not equal to the correct result. [18]	Integer arguments of arithmetic operations, such as signed Add32 or signed Mul32.	The arithmetic operation on the containers results in value smaller than the associated representation.	{x>0 AND y>0 AND x+y<0} (for containers of signed Add32) {x>0 AND y>0 AND x× y<0} (for containers of signed Mul32)
Incorrect conversion between numeric types	When converting from one data type to another, such as long to integer, data can be omitted or translated in a way that produces unexpected values.[19]	Numeric argument of width conversion operations, such as 32 to 8 or 32 to 16 width conversions.	The width conversion on the container causes unexpected data omission or translation.	{x>0×000000ff} (for containers of 32 to 8 width conversion) {x>0×0000ffff } (for containers of 32 to 16 width conversion)

The containers of vulnerabilities are specified based on the analysis level of the target program. For example, if the detection algorithm analyzes the C++ source code of the target program to detect vulnerabilities in it, the containers are specified according to the syntax of C++ language. For instance, to specify the integer overflow vulnerability in C++ source codes, the containers are defined as integer variables that are used in arithmetic operations. The detection algorithm searches in the source code for such variables. It then checks the satisfiability of the vulnerability rule for the located variables. In the next section we present a general vulnerability seeking algorithm for detecting the vulnerabilities that are represented in this model. Then, in Section 5, we present a method for specifying the single-instruction vulnerabilities to be detected in the executable codes and the details of employing the general vulnerability seeking algorithm for detecting the specified vulnerabilities in the executable codes. The presented vulnerability seeking algorithm analyzes the executable codes, locates the specified containers and check the satisfiablity of the relevant vulnerability rules for those containers.

Our general model can be enhanced to cover multi-instruction vulnerabilities too. Since multi-instruction vulnerabilities appear through a sequence of instructions in the program, they are specifiable through a sequence of pairs of Container(s), Rule. So the model should be enhanced to specify these vulnerabilities through a sequence of pairs of Container(s), Rule. A more detailed study of the issue is beyond the scope of this paper, as the main purpose of the paper is introducing the basic idea and suitability of Container, Rule model for describing and detecting software vulnerabilities. We have extended our vulnerability detection mechanism to detect heap-based and stack-based buffer overflow vulnerabilities, which are multi-instruction software vulnerabilities, and it achieved acceptable results in the experiments reported in [28,29]. Since these works are more technical, we have not included the related material in this paper.

4. A general extendable vulnerability seeking algorithm

Generally, our proposed vulnerability seeking algorithm searches for the containers of specified vulnerabilities in the program code. It compares each statement of the program code with the containers of the specified vulnerabilities. If there is a match, the algorithm checks the satisfiability of the relevant vulnerability rule for that statement. In other words, it searches for data values in that containers which make the vulnerability rule true.

Our proposed vulnerability seeking algorithm uses concolic (concrete + symbolic) execution technique to analyze the target program and detect the specified vulnerabilities in it. In this technique, the program code is instrumented and executed with concrete input data. During the execution, the constraints on input data in the executed path are calculated symbolically. These constraints, called the path constraints, determine the characteristics of input data that traverse the executed path in the program. After the execution with concrete input data, one of the calculated path constraints is negated and the satisfiability of the resulted set of constraints is queried from a SMT solver. SMT solvers receive a set of variables and a set of predicates on these variables and find values for the variables that satisfy the predicates or inform that the predicates are not satisfiable. If the SMT solver returns a solution that satisfies the new constraints, the solution is used to generate new concrete input data that traverse a different execution path in the program. Negating the calculated path constraints and generating new input data are repeated to analyze as many execution paths in the program as possible.

We consider generation of appropriate test data for detecting a vulnerability in a statement as solving a satisfaction problem. Due to the similarity between our view point and the concolic execution technique, our vulnerability seeking algorithm uses this technique to generate appropriate test data to detect the specified vulnerabilities in each execution paths of the program. Our algorithm calculates symbolic vulnerability constraints, in addition to the symbolic path constraints, for each execution path in the program. Symbolic vulnerability constraints determine the characteristics of input data that activate a specific vulnerability in the intended instruction of an execution path.[1] The vulnerability con-

[1] By "activating a vulnerability" we mean making the specified vulnerability active in the intended statement and causing a security error in the executed path.

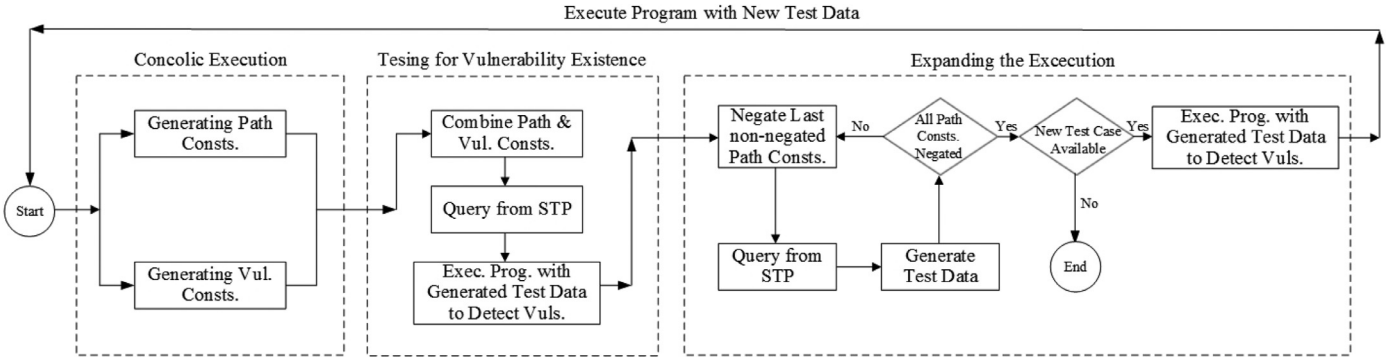

Fig. 2. Steps of the proposed vulnerability seeking algorithm.

straints are calculated for any statement that matches with the container of a specified vulnerability based on the relevant vulnerability rule.

The concolic execution technique, that was proposed by Godefroid in [31], has been used in various vulnerability seeking algorithms to increase their coverage on the target program code, such as [32] and [33]. For example, Dowser [33] is proposed to detect buffer overflows in executable codes. It uses the concolic execution technique to calculate the path constraints and generate appropriate test data that traverse new execution paths in the program. It also analyzes the executable code statically to locate the loops in the program. This information is used to explore the execution paths with complex loops with more priority. Considering vulnerability detection as a satisfaction problem and calculating the vulnerability constraints in addition to the path constraints during the concolic execution, has also been performed previously in some smart fuzzers. For example, Sage [34] and CATCHCONV [14] are two smart fuzzers that detect integer vulnerabilities in the executable codes by proving the satisfiability of the vulnerability constraints in the program. Also, EXE [13] and Klee [12] use the same method to detect buffer overflow in C and C++ codes. These fuzzers, however, do not formally model and specify the vulnerabilities and only detect one or a limited number of software vulnerabilities. On the contrary, our vulnerability seeking algorithm calculates the vulnerability constraints based on the rules of the specified vulnerabilities. In fact, our vulnerability seeking algorithm detects automatically the specified vulnerabilities by finding appropriate data that satisfy the rule of the vulnerabilities for the relevant statements of the program code.

4.1. The algorithm

Fig. 2 illustrates the main steps of our vulnerability seeking algorithm, i. e. generating the path and vulnerability constraints, testing for vulnerability existence and expanding the program execution. Each step is described in the following sections.

4.1.1. Generating the path and vulnerability constraints

In the first step, the program is executed with some random concrete input data. The program code is instrumented to monitor the flow of untrusted input data–tainted data– and calculate symbolic path and vulnerability constraints during the execution. Listing 1 presents the pseudo code of how the path and vulnerability constraints are calculated in our algorithm.

As shown in Listing 1, our algorithm checks whether each executed Vex statement is a jump instruction or matches with any of the containers of specified vulnerabilities. If it is a jump statement, function *generate_path_const()* is called to calculate a new path constraint based on that statement. This function extracts the condition of the jump instruction and generates a new path constraint accordingly. Since the goal of calculating the path constraints is to generate new input data that traverse other execution paths in the program, the path constraints are calculated for the jump instructions that depend on tainted data. In other words, only the conditions of the jump instructions that are affected by input data are calculated.

```
for(each st in the executed statements)
 {
   if(matches(st, jmp_statement))
       generate_new_path_constraint();
   for(each cont_i in CONTs)
   {
      if(matches(st,cont_i)
        generate_vul_const(st,vul_i);
   }
 }

generate_vul_const(st,vul_i)
{
    switch(vul_i){
         case INT_Ovf:
              generate_int_overflow_vul_const(st);
         case INT_Unf:
               generate_int_underflow_vul_const(st);
         case DIV_BY_ZERO:
               generate_div_by_zero_vul_const(st);
         ......
    }
}
```

Listing. 1. Pseudo code of calculation of path and vulnerability constraints in the proposed algorithm.

For the statements that match with the container of a specific vulnerability, a function that calculates the relevant symbolic vulnerability constraints for that statements is called. The vulnerability constraints are calculated according to the specified rule for that vulnerability.

Since the vulnerabilities are usually exploited by malicious input data, we generate symbolic vulnerability constraints for the statements that have tainted containers. Actually, it is possible to calculate the vulnerability constraints for all the containers regardless of being tainted or not. However, in this way, we need to monitor all the data flows in the program and generate a large number of vulnerability constraints for various statements while only a few of them are exploitable.

The security error might not be handled appropriately and result in compromising the security policy [30].

```
Query=true;
for(i=0;i<num_of_consts;i++){
    if(type(const_i)==VUL_CONST)
            Query=const_i;
            for(j=i;j>=0;j--)
            {
                if(type(const_j)==PATH_CONST)
                    Query=(Query AND const_j);
            }
}
```

The combined constraints are queried from a SMT solver, called STP.

Listing. 2. Combining the path and vulnerability constraints.

4.1.2. Testing for vulnerability existence

At the second step, each vulnerability constraint is combined with its previous path constraints. The combined constraints determine the characteristics of input data that traverse the same execution path, reach the intended statement and activate a specific vulnerability in that statement. Combining the path and vulnerability constraints is performed by ANDing these constraints. Listing 2 presents the pseudo code of this step.

The combined constraints are queried from a SMT solver, called STP.

4.1.3. Expanding the execution

At the third step, the algorithm generates new input data to expand the execution into new paths. We use the same method as in [14,32] and [34] to generate new input data that traverse other execution paths. In this step, the calculated path constraints for the executed path are negated one by one, from the last to the first. After each negation, the new set of constraints are queried from STP. If STP returns a solution that satisfies the constraints, it is used to generate new concrete input data. The generated data are used to restart the algorithm from the first step to traverse other execution paths in the test program and detect the vulnerabilities in those paths.

4.2. Extending the algorithm

As Fig. 2 shows, the general steps of our vulnerability seeking algorithm do not depend on a specific vulnerability. At the first step, the algorithm calculates the vulnerability constraint for the statements that match with the containers of the specified vulnerabilities. These constraints are calculated based on the rule of the relevant vulnerability. Satisfiability of the calculated constraints is decided at the second step and appropriate test data are generated to detect specific vulnerabilities. The algorithm can be extended to detect a new vulnerability with inserting the containers of the vulnerability to the set of vulnerability containers and adding the relevant vulnerability constraint calculation routines to the function *generate_vul_const()*.

5. An extendable vulnerability detection method for executable codes

As mentioned before, an extendable vulnerability detection method requires a general vulnerability specification method and a vulnerability seeking algorithm that understands specified vulnerabilities. In this section, we present such an extendable method for detecting the vulnerabilities in executable codes.

When analyzing the executable codes, it is helpful to translate the assembly instructions into an intermediate representation. Such translation presents more information about the instructions and their side effects. It also helps to access each instruction and its arguments programmatically. This benefit inclined us to use such an intermediate representation in specifying the vulnerabilities and designing the detection method. We designed and implemented our method using Valgrind. Valgrind is a framework for instrumenting executable codes and implementing dynamic analysis solutions [20]. We specify vulnerabilities based on the intermediate language presented in Valgrind, called Vex [35]. Also, the vulnerability seeking algorithm is implemented as a plug-in for Valgrind. The specification method and vulnerability seeking algorithm are described in the following sections.

5.1. Vulnerability specification method

As vulnerabilities are specified based on the intermediate language Vex, it is necessary to have a short introduction on it. This helps the reader to understand the suggested method better. An interested reader is referred to [35] for more information.

5.1.1. The Vex language

Valgrind translates each assembly instruction into one or more statements in Vex language. As an example, consider the following assembly instruction:

```
addl %eax, %ebx
```

Translating the above instruction into Vex results in the following statements:

```
t3 = GET:I32(0)
# get %eax, a 32-bit integer
t2 = GET:I32(12)
# get %ebx, a 32-bit integer
t1 = Add32(t3,t2)
# addl
PUT(0) = t1
# put %eax
```

An assembly code that is translated into Vex consists of a set of code blocks. Each code block contains around one to fifty instructions. An instruction is defined in a data structure called statement. Statements have different types, e. g. WrTmp, Put, Store, etc. The following is an example WrTmp statement and its equal assembly code. This statement means that the value of EBX register is written into a temporary variable.

```
t2 = GET:I32(12)
# get %ebx, a 32-bit integer
```

The data structure of WrTmp statement is defined as follows in Vex.

```
Struct{
IRTemp tmp;
/* Temporary (LHS of assignment) */
IRExpr* data;
/* Expression (RHS of assignment) */
} WrTmp;
```

As you see this statement consists of two entities: one is a temporary value that defines where to write, which is t2 in this example. The other one is an expression that defines what to write in the temporary value.

The statements may contain one or more expressions. Expressions are defined in a different data structure. Like the statements, expressions have different types. For example, the statement in the previous example has an expression with type Get. Also, the type of the expression in the following WrTmp statement is binary Add32.

```
t1 = Add32(t3,t2)
```

Fig. 3 presents the structure of some of statements and expressions in Vex. We mentioned the common statements and expressions in this figure and refer the interested reader to [35] for the structure of other statements and expressions. As shown in this figure, the statements might contain one or more expressions.

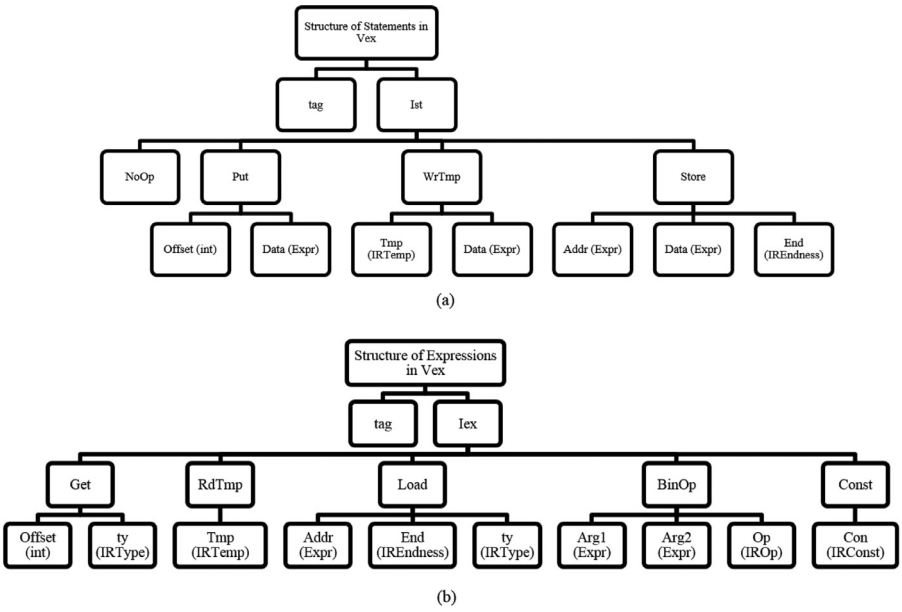

Fig. 3. The structure of statements and expressions in Vex. Part (a) shows the structure of some of the statements in Vex. Part (b) shows the structure of some of the expressions in Vex.

The expressions also might consist of different expressions. Various data types are defined in Vex for the elements of statements and expressions. For example, IRType in Get statement defines the type of the value that is read from the register. IREndness in the Store statement defines the endian-ness of the store operation. Also, IROp in the BinOp expression defines the type of binary operation.

5.1.2. Specification based on Vex

We use the structure of the statements in Vex for specifying the containers of single-instruction vulnerabilities. The containers define the data entities in specific instructions that should be checked against the vulnerability rule. Thus, the containers are defined as the data elements of a specific statement in Vex. For example, integer overflow may occur in an Add32 operation. This operation is represented in Vex with the following example statement:

```
ti=Add32:I32(tx,ty)
```

In this statement, *tx* and *ty* are the containers for integer overflow vulnerability. Fig. 4 shows the structure that defines these containers. The Not_Important value in this figure is assigned to the parts of the statement that are not important in the specific vulnerability.

As mentioned before, the rule is defined as a first-order formula on the containers. Using this method, the sample vulnerabilities division by zero, NULL pointer dereference and integer bugs are specified as follows.

Division by zero

Division operation exists in WrTmp statements with expressions like DivS32, DivU32, etc. If in such statements the divisor argument is equal to zero, the program may be vulnerable. Thus, the container for this vulnerability is defined as the divisor argument of the division expression in a WrTmp statement. Using the data structure of statements, a container is defined as the following for DivS32 expression.

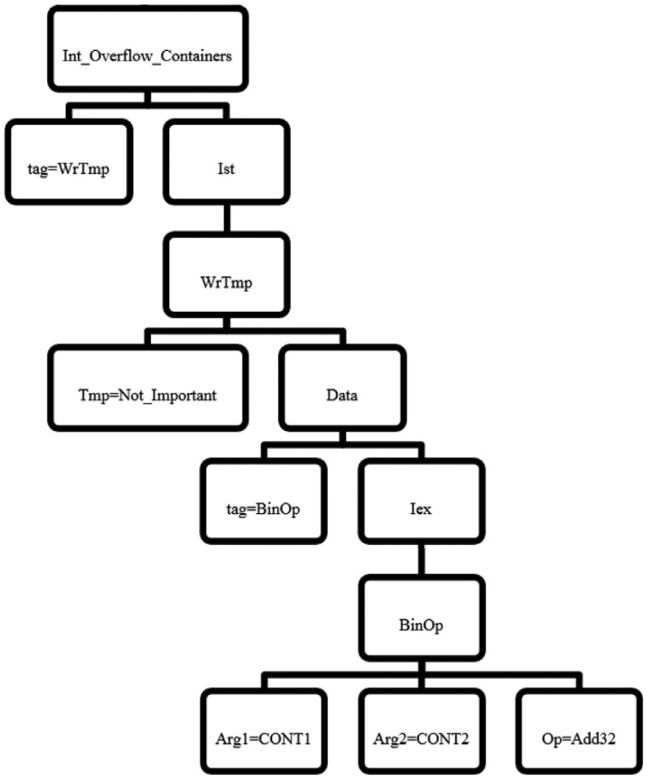

Fig. 4. The structure that defines two containers for integer overflow.

```
Container[DIV_BY_ZERO].tag = Ist_WrTmp;
Container[DIV_BY_ZERO].WrTmp.tmp = Not_Important;
Container[DIV_BY_ZERO].WrTmp.data->tag= Iex_Binop;
Container[DIV_BY_ZERO].WrTmp.data->Iex.Binop.op = Iop_DivS32;
Container[DIV_BY_ZERO].WrTmp.data->Iex.Binop.arg1=CONT;
Container[DIV_BY_ZERO].WrTmp.data->Iex.Binop.arg2 = NULL;
```

Note that Not_Important and NULL values are assigned to the parts of the statement that are not important in this vulnerabil-

ity. By assigning the constant value CONT to the first argument of binary operation, the container is determined to be the first argument of this operation. If the value of the container is equal to zero, the program may be vulnerable. To be consistent with specification of the containers, the rules are defined using the Vex language too. In other words, the formula on the containers are represented using the expressions defined in Vex. For division by zero vulnerability the rule is represented as follows:

```
if (CmpEQ8(CONT,0x00:I8))
```

In this rule, the operator CmpEQ8 is one of the comparison operators defined in Vex.

NULL pointer dereference

When a program dereferences a pointer with NULL value it may be vulnerable to NULL pointer dereferences [21]. Pointers are dereferenced with the store statements and the load expressions in Vex. For example, in the below statements t21 and t25 are the pointers for memory write and read operations respectively. If any of them is equal to zero, the program may be vulnerable.

```
STle(t21) = t5
t27 = LDle:I32(t25)
```

Thus, the container for this vulnerability is the address argument of the load expression and store statement. The following containers are defined for this vulnerability.

```
Container[NULL_D1].tag= Ist_WrTmp;
Container[NULL_D1].WrTmp.tmp =Not_Important;
Container[NULL_D1].WrTmp.data->tag= Iex_Load;
Container[NULL_D1].WrTmp.data->Iex.Load.addr= CONT;
Container[NULL_D1].WrTmp.data->Iex.Load.end= Not_Important;
Container[NULL_D1].WrTmp.data->Iex.Load.ty=Not_Important;
```

The above container is defined for a Load expression in a WrTmp statement. The container for a store statement is defined as follows:

```
Container[NULL_D2].tag= Ist_Store;
Container[NULL_D2].Store.addr= CONT;
Container[NULL_D2].Store.data= NULL;
Container[NULL_D2].Store.end=Not_Important;
```

The rule here is similar to the rule of division by zero. If any of the containers is equal to zero the program may be vulnerable. Thus, the rule is defined as follows;

```
if (CmpEQ8(CONT,0x00:I8))
```

Integer bugs

Integer vulnerabilities were studied by Molnar et al. in [14] in 2009. They classified integer vulnerabilities into three classes: integer overflow/underflow, width conversion errors and signed/unsigned conversion errors. A number of vulnerability constraints were also suggested for each class. We have used the same classification, but redefined the vulnerability constraints and added new constraints to specify the rule of integer vulnerabilities. Since the focus of this paper is on single-instruction vulnerabilities, signed/unsigned conversion errors that appear in more than one instruction are not covered here. Table 2 presents our specifications of the covered integer bugs.

To prepare the specifications of Table 2, we first collected possible containers for these vulnerabilities based on their definitions in CWE. The arguments of 32-bit arithmetic operations, e.g. Add32 or Sub32, are considered as the containers of the 32-bit integer bugs.[2] We then analyzed each arithmetic operation to find out which container(s) can cause a specific integer bug in that operation. Based on this analysis, the rules are generated for the containers. By applying such a systematic approach in specifying these integer bugs, a more comprehensive specification is presented in this paper. Also, as it will be explained in Section 6, our specifications are more efficient than the ones presented in [14] in detecting the vulnerabilities in the benchmark programs.

The containers for these integer bugs are generally the arguments of a WrTmp statement. The operator in this statement may be a binary operation or a unary one. For example, the containers for Add32 binary operation are as follows:

```
Container[INT_OVF].tag = Ist_WrTmp;
Container[INT_OVF].WrTmp.tmp = Not_Important;
Container[INT_OVF].WrTmp.data->tag= Iex_Binop;
Container[INT_OVF].WrTmp.data->Iex.Binop.op = Iop_Add32;
Container[INT_OVF].WrTmp.data->Iex.Binop.arg1= CONT1;
Container[INT_OVF].WrTmp.data->Iex.Binop.arg2 = CONT2;
```

Also, the container for a WrTmp statement with Iop_32to8 unary operation is defined as follows:

```
Container[Width-Conv].tag = Ist_WrTmp;
Container[Width-Conv].WrTmp.tmp = Not_Important;
Container[Width-Conv].WrTmp.data->tag= Iex_Unop;
Container[Width-Conv].WrTmp.data->Iex.Binop.op = Iop_32to8;
Container[Width-Conv].WrTmp.data->Iex.Unop.arg= CONT;
```

In Table 2, the column Container Operator presents only the operator of the WrTmp statement because of lack of space. The description column in this table presents a short description for the first order formula in each specification.

We explain some of the specifications in more details here for a better understanding. For example, an integer overflow happens for a Signed Add32 operator when addition of two negative integer data results in a positive value. Thus, the rule first checks if the containers can be negative. This is done by using a signed comparison operator, i. e. CmpLT32S. If the first two constraints are true, the third constraint is checked. In the third constraint it is checked whether the result of Add32 is positive or not. This is done by comparing the unsigned value of the result with 0×80000000 using an unsigned comparison operator, i. e. CmpLT32U. If the unsigned result is less than 0×80000000, it means that the left-most bit of the result is 0 and therefore it is positive.

Also, an underflow occurs in Signed Add32 when addition of two positive integer data results in a negative value. Thus, the rule first checks if the containers are positive and the unsigned value of the result is greater than $0\times7fffffff$. When the left-most bit in the result is equal to 1, its unsigned value would be greater than $0\times7fffffff$. In other words, when the result is negative, it would be greater than $0\times7fffffff$ in an unsigned comparison.

The containers related to the operators Unsigned Mul32 and Unsigned Add32 contain unsigned values. Thus, it is not possible to detect the bug by checking the sign of the result. Therefore, the result is compared with the 64-bit value of $0\times00000000ffffffff$. If the result of Unsigned Add32 or Unsigned Mul32 is greater than this value, an overflow has occurred. In order to prevent from getting wrap-around, the containers are first extended into 64 bits and the addition and multiplication is performed using Add64 and Mul64. In this way, the overflowed bit appears in the 32nd left-most bit of the 64-bit result.

5.2. The vulnerability seeking algorithm

Section 4 presented the general vulnerability seeking algorithm that detects the specified vulnerabilities in the program. In this section, we describe the design and implementation details of

[2] We only consider these bugs for the 32-bit integer type.

Table 2
Integer bugs specification.

Vulnerability	Container operator	Rule	Description
Integer overflow	Unsigned Add32	if CmpLT64U(0×00000000ffffffff:I64 ,Add64(32Uto64(CONT1), 32Uto64(CONT2))))	X1+X2 > 0×00000000ffffffff
	Unsigned Mul32	If (cmpLT64U(0×00000000ffffffff:I64, MulU64(32Uto64(CONT1), 32Uto64(CONT2)))	X1∗X2 > 0×00000000ffffffff
	Signed Add32	if CmpLT32S(CONT1,0×00:I32)) AND if (CmpLT32S(CONT2,0×00:I32)) AND if CmpLT32U(Add32(CONT1,CONT2), 0×80000000:I32))	X1< 0, X2< 0, X1+X2> 0
	Signed Mul32	if CmpLT32S(CONT1,0×00:I32)) AND if (CmpLT32S(0×00:I32,CONT2)) AND if (CmpLT32U(Mul32(CONT2,CONT1), 0×80000000:I32))	X1< 0, X2> 0, X1∗X2> 0
		if (CmpLT32S(0×00:I32,CONT1)) AND if (CmpLT32S(CONT2,0×00:I32)) AND if (CmpLT32U(Mul32(CONT2,CONT1), 0×80000000:I32))	X1> 0, X2< 0, X1∗X2> 0
	Sub 32	if (CmpLT32S(CONT1,0×00:I32)) AND if (CmpLT32S(0×00:I32,CONT2)) AND if (CmpLT32U(Sub32(CONT1,CONT2), 0×80000000:I32))	X1< 0 , X2> 0, X1-X2> 0
Integer underflow	Signed Add32	if CmpLT32S(0×00:I32,CONT1)) AND if (CmpLT32S(0×00:I32,CONT2)) AND if (CmpLT32U(0×7fffffff:I32, Add32(CONT1,CONT2)))	X1> 0, X2> 0, X1+X2 < 0
	Signed Mul32	if (CmpLT32S(0×00:I32,CONT1)) AND if (CmpLT32S(0×00:I32,CONT2)) AND if (CmpLT32U(0×7fffffff:I32, Mul32(CONT2,CONT1)))	X1> 0, X2> 0, X1∗X2< 0
	Sub32	if (CmpLT32S(0×00:I32,CONT1)) AND if (CmpLT32S(CONT2,0×00:I32)) AND if (CmpLT32U(0×7fffffff:I32, Sub32(CONT1,CONT2)))	X1> 0, X2< 0, X1-X2< 0
Width conversion	Iop_32to8	if CmpLT32U(0×000000ff:I32,CONT)	X> 0×000000ff
	Iop_32to16	if CmpLT32U(0×0000ffff:I32,CONT)	X> 0×0000ffff
	Iop_Not32	if CmpNE32(CONT,0×80000000:I32))	X! =0×80000000
	Get8	if CmpLT32U(0×000000ff:I32,GET:I32(CONT))	X> 0×000000ff
	Get16	if CmpLT32U(0×0000ffff:I32,GET:I32(CONT)))	X> 0×0000ffff

our vulnerability seeking algorithm for executable codes. As mentioned before, we have implemented our algorithm as a plug-in for Valgrind. Since Valgrind translates the executable codes into Vex language, we use Vex to specify the vulnerabilities in the {Container(s), Rule} structure. The implemented vulnerability seeking algorithm instruments the Vex statements of the program code to perform taint analysis and calculate the path and vulnerability constraints at the first step of Fig. 4.

In order to calculate the path constraint, our vulnerability seeking algorithm instruments the jump statements, that are called Exit statements in Vex, of the program code. If the condition of the jump depends on tainted data, a new symbolic path constraint is calculated by function *generate_path_const()*. For example, in the following C code, there are two *if* statements before the *printf()* function. Each *if* statements is translated into a jump instruction in the equivalent executable codes. One of the jump instructions depends on a tainted variable and thus one path constraint is generated for the execution path that contains the *printf()* function.

```
void MySub()
{
        char inputBuffer[10]="", mychr;
        int pathTrue=1;
        fgets(inputBuffer, 10, stdin);
        mychr= inputBuffer[0];
        if (pathTrue==1)
        {
            if (mychr=='a')
                printf("The input starts with a");
        }
}
=> Generated path constraint:  if (CmpEQ8(32to8(8Uto32(GET:I8(PUT(8Uto32(
                                  LDle:I8(input(0)))))),0x61:I8))
```

Our algorithm monitors the flow of tainted data in the program during the program execution and records the sequence of executed operations on each tainted byte. Thus, when a new path constraint is calculated for a tainted byte in a jump instruction, we know how the input data has changed from the beginning until that instruction. The above constraint shows what operations are performed on the tainted byte *input(0)* until it reaches the *if* statement. As our algorithm instruments the equivalent Vex representation of the program, the operations shown in the constraint are expressed with the operators of Vex language.

In order to calculate the vulnerability constraints, each Vex statement in the program code is compared with the containers of the specified vulnerabilities. For the statements that matches with the containers of a specified vulnerability, a new symbolic vulnerability constraint is calculated by the relevant function. For example, Listing 3 presents the pseudo code of *generate_div_by_zero_vul_const* function that generates the constraint for the division by zero vulnerability. This function generates a new vulnerability constraint to check whether the 8-bit tainted container of a division operation can be zero.

As the flow of tainted data is monitored during the program execution, we record the sequence of executed statements on each byte of tainted data. Thus, we know what tainted byte is currently in *st.WrTmp.data->Iex.Binop.arg1* and what operations have been performed on it from the beginning.

After calculating the path and vulnerability constraints, according to Fig. 4, the constraints are combined and solved to generate new test data that reach a specific Vex statement in the executed path and activate a specific vulnerability in that statement. As Listing 2 shows, each vulnerability constraint is merged with its previous path constraints at the second step of our algorithm. For example, the following shows part of a C program that contains the division by zero vulnerability and the calculated vulnerability constraint by our algorithm. Since the equivalent Vex representation of this code is long, we just present the C code here:

```
scanf("%s", buffer);
int z=2/((int)buffer[1]+(int)buffer[2]);

Generated vulnerability constraint =>
if (CmpEQ32(Add32(LDle:I32(STle(8Sto32(GET:I8(PUT(8Uto32(LDle:I8(input(1))))))),
 LDle:I32(STle(8Sto32(GET:I8(PUT(8Uto32(LDle:I8(input(2))))))),0x0:I32))
```

Our vulnerability seeking algorithm queries the combined constraints from STP. STP is a constraint solver that decides the satisfiability of first-order logic formulas over *bitvector* and array terms. A bitvector is an array of Boolean variables. The input of STP consists of the definition of variables, a set of assertions on defined variables and a query about the status of some variables. STP determines if the query is satisfiable and presents a counter-example otherwise. As an example, the following shows a sample input to STP that defines two 8-bit terms *x* and *y*, and makes a query about them to check whether x∗y is equivalent to y∗x and x is not less than y.

```
void generate_div_by_zero_vul_const(st)
{
    if (is_tainted(st.WrTmp.data->Iex.Binop.arg1))
        printf("if (CmpEQ8(%s,0x00:I8)", st.WrTmp.data->Iex.Binop.arg1) ;
}
```

Listing. 3. Pseudo code of function generate_div_by_zero_vul_const.

Table 3
Sample constraint translation into STP query.

C code	`int main(int argc, char *argv[]){ int fd,result; char buffer[100]; fd = open(argv[1], O_RDONLY); read(fd, buffer, 1000); result=12/(int)buffer[1]+(int)buffer[2]; }`
Generated vulnerability constraint	if (CmpEQ32(Add32(LDle:I32(STle(8Sto32(GET:I8(PUT(8Uto32 (LDle:I8(input(1))))))),LDle:I32(STle(8Sto32(GET:I8(PUT(8Uto32 (LDle:I8(input(2))))))),0x0:I32))
STP Query	`x1 : BITVECTOR(8); x2 : BITVECTOR(8); QUERY (((BVPLUS(32,BVSX(x1,32), BVSX(x2,32)) = 0h00000000)));`

```
x, y : BITVECTOR(8);
ASSERT(x=0hex05);
ASSERT(y = 0bin00000101);
QUERY(
BVMULT(8,x,y)=BVMULT(8,y,x)
AND
NOT(BVLT(x,y))
);
```

To be consistent with the syntax of STP queries, we first translate the combined constraints into the appropriate language, i. e. SMT-Lib2. The SMT-Lib2 language, that is used to express the STP queries, contains equivalent operations for most of the logical and arithmetic operations in the executable codes. For example, in the above example, BVMULT represents the multiplication operation on variables *x* and *y*.

In order to represent the calculated constraints in SMT-Lib2 format, the Vex operators in the constraints are represented in their equivalent SMT-Lib2 format. As an example, Table 3 presents a sample C program that contains a division by zero vulnerability, the calculated vulnerability constraint and its equivalent STP query. In this translation, the load, store, get and put statements are ignored as they are used by the intermediate statements and have no effects on the final constraint. The 8Sto32 Vex operator in the vulnerability constraint is translated into BVSX that is the signed length extension operator in SMT-Lib2. The resulted STP query asks if there are two 8-bit variables that their signed 32-bit addition results in zero. If possible, STP returns two 8-bit values, for *x1* and *x2*, that satisfy the query. These values are used as the first and second bytes of the new test input data to cause division by zero in the intended instruction. The program is executed with the new test data and if the data cause undefined or unacceptable behavior in the program, a new vulnerability would be reported by the algorithm.

6. Experiments

We have implemented our proposed method as a plug-in for Valgrind. Since our vulnerability seeking algorithm uses the concolic execution technique, we have used a plug-in of Valgrind, called Fuzzgrind, that performs concolic execution on Vex statements and generates STP queries for the calculated path constraints [32]. In fact, we have implemented our vulnerability seeking algorithm by extending Fuzzgrind so that it calculates the specified vulnerability constraints in addition to the path constraints and generates appropriate STP queries to detect vulnerabilities in each execution path. The implemented algorithm is tested in a Backtrack VMware with 1 GB RAM and 1.8 GHz CPU.

The implemented vulnerability seeking algorithm is tested on Juliet_Suite_v1.2_for_C_Cpp benchmark of NIST SAMATE test case collections. The goal of these experiments is to verify if the specified vulnerabilities are accurate and if the implemented vulnerability seeking algorithm is able to detect these vulnerabilities. The chosen benchmark consists of different vulnerable programs for a number of vulnerabilities defined in CWE. We have tested our algorithm on four groups of vulnerable programs: CWE190_Integer_Overflow, CWE191_Integer_Underflow, CWE369_Divide_by_Zero and CWE476 _NULL_Pointer_Dereference.

The test programs contain one or more good functions and usually a bad function. Good functions avoid a vulnerability by checking the (input) data value or changing it to a fix value before using in critical operations. The bad function operates on input data directly with no previous checks. For example, a bad function in a test program that is vulnerable to integer overflow is as follows:

```
void CWE190_Integer_Overflow__char_fscanf_add_01_bad()
{
    char data;
    data = ' ';
    fscanf (stdin, "%c", &data);
    {
    char result = data + 1;
    printHexCharLine(result);
  }
}
```

Some test programs read the input data from a network socket, an input file or the keyboard. Some of them operate on fixed random values that are defined in the code. Since our implemented taint analysis method considers only the input data from keyboard and input files as tainted, we did not test the programs that get the input data from sockets. Moreover, since fixed data values are not usable in exploiting the program by malicious users, our tests do not cover the programs that become vulnerable by manipulating such data.

For each test program, our algorithm generates a number of test-cases based on the calculated path and vulnerability constraints. For each path, the program is executed with the test-cases that are generated with solving the vulnerability constraints. If the program crashes or shows any pre-defined bad behavior during the execution, our algorithm reports the related vulnerability.

Tables 4–7 present the results of testing our algorithm with some of the programs in CWE190_Integer_Overflow and CWE191_Integer_Underflow groups. For the matter of space, only a limited number of test results are presented in these tables. The other test programs are similar to the programs that are listed in these tables and testing them achieved similar results.

The columns in Table 4 present, from left to right, the name of test program, number of vulnerabilities in it, the number of calculated vulnerability constraints for containers with width con-

Table 4
Integer overflow addition test results. Names of the test programs are shorten because of lack of space. The complete name of each test program is the result of appending its name in the table to "CWE190_Integer_Overflow_char_fscanf".

Sample name	# Vulns	# Width- Conv	# S Add32	# U Add32	# Test-cases	# Vul test-cases	# TP	# FP	# TN	# FN	Time (s)
_add_01	1	2	2	2	6	4	1	0	2	0	2.43
_add_11	1	5	5	5	7	5	1	0	4	0	2.89
_add_21	1	5	5	5	3	2	1	0	3	0	2.91
_add_31	1	3	3	3	5	3	1	0	2	0	2.68
_add_41	1	3	3	3	4	3	1	0	2	0	2.8
_add_51	1	5	5	5	3	2	1	0	2	0	2.95
_add_61	1	3	3	3	4	3	1	0	2	0	2.58

Table 5
Integer overflow multiply test results. Names of the test programs are shorten because of lack of space. The complete name of each test program is the result of appending its name in the table to "CWE190_Integer_Overflow_int_fgets".

Sample name	# Vulns	# Mul32	# Test-cases	# Vul test-cases	# TP	# FP	# TN	# FN	Time (s)
_multiply_01	1	8	7	2	1	0	2	0	8.99
_multiply_10	1	18	9	3	1	0	4	0	10.2
_multiply_21	1	10	7	2	1	0	3	0	9.41
_multiply_31	1	8	7	2	1	0	3	0	8.38
_multiply_41	1	6	7	2	1	0	2	0	8.39
_multiply_51	1	6	7	2	1	0	2	0	8.43
_multiply_61	1	6	11	3	1	0	2	0	10.5

Table 6
Integer underflow subtraction test results. Names of the test programs are shorten because of lack of space. The complete name of each test program is the result of appending its name in the table to "CWE191_Integer_Underflow__char_fscanf".

Sample name	# Vulns	#Width -conv	# sub32	# Test-cases	# Vul test-cases	# TP	# FP	# TN	# FN	Time (s)
_sub_01	1	3	3	4	3	1	0	2	0	2.52
_sub_10	1	3	4	2	1	1	0	4	0	2.64
_sub_21	1	3	4	4	1	1	0	3	0	2.6
_sub_31	1	3	3	4	3	1	0	2	0	2.53
_sub_41	1	3	3	4	3	1	0	2	0	2.51
_sub_51	1	3	3	4	3	1	0	2	0	2.52
_sub_61	1	3	3	4	3	1	0	2	0	2.46

Table 7
Integer underflow multiply test results. Names of the test programs are shorten because of lack of space. The complete name of each test program is the result of appending its name in the table to "CWE191_Integer_Underflow_int_fgets".

Sample name	# Vulns	# Mul32	# Test-cases	# Vul test-cases	# TP	# FP	# TN	# FN	Time (s)
_multiply_01	1	6	7	2	1	0	2	0	9.15
_multiply_10	1	8	6	1	1	0	4	0	13.1
_multiply_21	1	8	6	1	1	0	3	0	10.0
_multiply_31	1	6	9	2	1	0	2	0	11.8
_multiply_41	1	6	7	2	1	0	2	0	9.04
_multiply_51	1	6	7	2	1	0	2	0	8.94
_multiply_61	1	6	9	2	1	0	2	0	11.9

version, Signed Add32 and Unsigned Add32 operators, the number of generated test-cases (that are generated with solving path and vulnerability constraints), the number of generated test-cases with solving vulnerability constraints, the number of TP (True Positive), FP (False Positive), TN (True Negative) and FN (False Negative) in the test results and the duration of performing the test. Other tables have similar columns except for the columns of the number of calculated vulnerability constraints. These columns change depending on the test program to the number of calculated vulnerability constraints for the containers with Mul32 and Sub32 operators.

Note that the number of constraints are calculated for the whole testing process, that may include one or more execution of the test program. The number of generated test-cases demonstrates the number of solved path and vulnerability constraints. As shown in these tables, our implemented algorithm is able to calculate the vulnerability constraints for specified vulnerabilities and detect the specified vulnerabilities in the test programs.

We have also compared the accuracy of our specified vulnerabilities with the specified vulnerabilities in CATCHCONV, that is proposed by Molnar et al. [14]. CATCHCONV is a plug-in for Valgrind that uses concolic execution to detect integer bugs in executable codes. In this test, we first tested our implemented vulnerability seeking algorithm with the vulnerability specifications presented in CATCHCONV. Then, we repeated the test with our implemented algorithm using our vulnerability specifications. Figs. 5 and 6 present the results of comparing the vulnerability specifications presented in CATCHCONV and our vulnerability specifications. The graph shown in Fig. 5 compares the precision of the two specifications in detecting vulnerabilities in different groups of test programs. In each group of test programs, integer overflow occurs by manipulating an input value of a specific data type, such as int, char, unsigned int. There are 81 programs in each group that have the same vulnerability in an execution path with different complexities. The y axis in this graph represents the precision of the reported vulnerabilities by each specification, that is calcu-

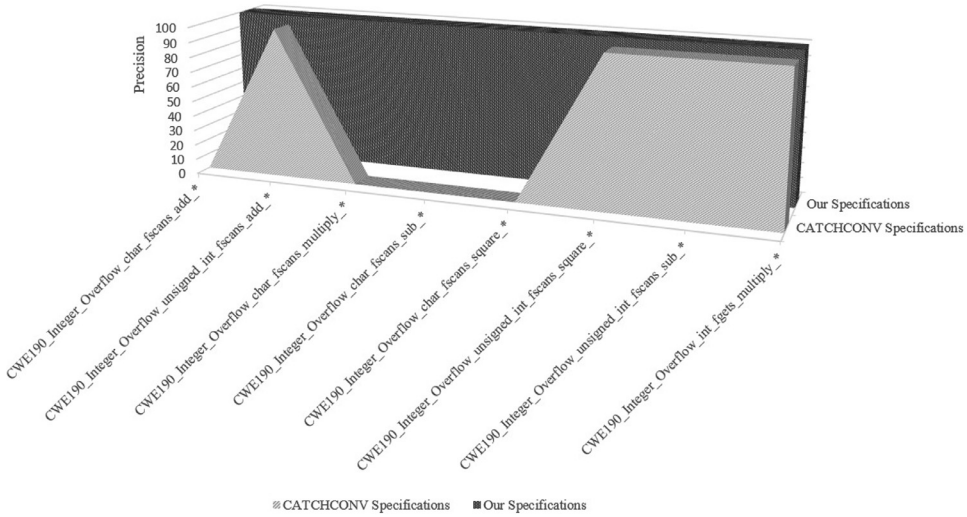

Fig. 5. Comparing the precision of vulnerability specifications of CATCHCONV and our vulnerability specifications.

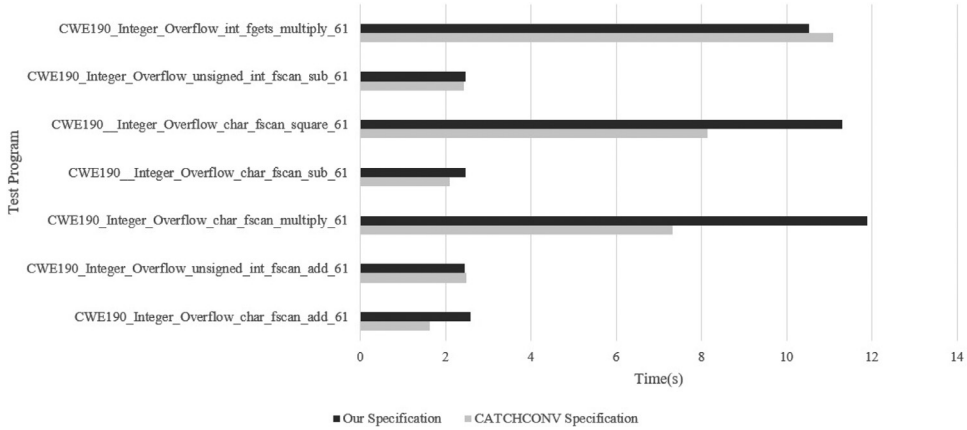

Fig. 6. Comparing the duration of tesing the benchmark programs with vulnerability specifications of CATCHCONV and our vulnerability specifications.

lated as the number of true positives over the total number of true positives and false positives [36]. Since each program contains one vulnerability, reaching the 100% precision for a group of programs means that the algorithm could detect correctly the vulnerabilities in all the programs of that group.

As shown in Fig. 5, our specified vulnerability constraints helped to detect the vulnerabilities in all these programs and achieved 100% precision for all the groups. However, the vulnerability specifications presented in CATCHCONV could not help to detect the vulnerability in the programs that manipulate the *char* data type and achieved 0% precision on these groups. It is because these test programs perform arithmetic operations on an 8-bit input character. However, the compilers translate these addition and subtraction operations to be performed with Add32 and Sub32 operators. In other words, these programs widen the input character implicitly to 32 bits and then add (or subtract) them with Add32 or Sub32 operators. Thus, the 8-bit character input can never make such 32-bit operations to overflow or underflow. In fact, the vulnerability in these programs appears when the result of the arithmetic operation is shortened into 8 bits in order to be stored in the 8-bit character variable. This is done by the use of GET8 expression in Vex, that extracts 8 bits of a 32-bit register. Since we have the last two vulnerability rules for width conversion in Table 2 of our specifications, our implemented algorithm can detect the vulnerabilities in these programs.

The graph of Fig. 6 compares the duration of testing a specific program in each group with our specifications and the specifications presented in CATCHCONV. As shown in this graph, the durations of testing the programs with the two specifications are rather equal, except for the programs that manipulate the *char* data type. This is because when we used our specifications, more vulnerability constraints were calculated and queried from STP for these programs. As a result, more test data were generated and executed by these test programs. Therefore, it took more time to test these programs when we used our specification. When we used the CATHCONV specifications, fewer vulnerability constraints were calculated for these programs that were not satisfiable. Thus, the testing process completed more quickly, albeit with false negative reports.

Table 8 illustrates the results of testing the algorithm on some programs in CWE369_Divide_by_Zero. In the bad function of these programs the input data is used as a divisor in a division or a modulo operation. If the divisor argument in such statement becomes zero, the program would crash. As shown in this table, our implemented algorithm could calculate the vulnerability constraints based on the specification of the division by zero vulnerability and detect the vulnerabilities in these programs. Finally, Table 9 presents the results of testing the algorithm on some of the programs in CWE476_NULL_Pointer_Dereference. In these test programs, a fix NULL value is explicitly assigned to a pointer and it causes a crash. We changed it into a more difficult scenario and

Table 8
Division by zero test results. Names of the test programs are shorten because of lack of space. The complete name of each test program is the result of appending its name in the table to "CWE369_Divide_by_Zero__int_fgets".

Sample name	# Vulns	# Div-by- zero	# Test-cases	# Vul test-cases	# TP	# FP	# TN	# FN	Time (s)
_divide_01	1	3	2	1	1	0	1	0	3.91
_divide_10	1	4	2	1	1	0	4	0	4.29
_divide_21	1	3	2	1	1	0	3	0	3.8
_divide_31	1	3	2	1	1	0	2	0	3.78
_divide_41	1	3	2	1	1	0	2	0	3.8
_divide_51	1	3	2	1	1	0	2	0	3.87
_modulo_01	1	3	2	1	1	0	2	0	3.15
_modulo_10	1	4	2	1	1	0	4	0	3.3
_modulo_21	1	4	2	1	1	0	3	0	3.41
_modulo_31	1	3	2	1	1	0	2	0	3.11
_modulo_41	1	3	2	1	1	0	2	0	3.17
_modulo_51	1	3	2	1	1	0	2	0	3.75

Table 9
NULL pointer dereferences test results.

Sample name	# Vulns	# Crashes	# NULL	# TP	# FP	# TN	# FN	Time (s)
CWE476_NULL_Pointer_Dereference__char_01	1	1	1	1	0	1	0	2.31
CWE476_NULL_Pointer_Dereference__char_10	1	1	1	1	0	4	0	2.86
CWE476_NULL_Pointer_Dereference__char_21	1	1	1	1	0	3	0	2.57
CWE476_NULL_Pointer_Dereference__char_31	1	1	1	1	0	1	0	2.38
CWE476_NULL_Pointer_Dereference__char_41	1	1	1	1	0	1	0	2.41

made the pointer dependent on the input data. The implemented algorithm could detect both scenarios. For the first scenario, the algorithm is changed so that it does not care about the performance and generates the constraints for all containers regardless of being tainted or not. For the second scenario, it creates vulnerability constraints only for tainted containers. Based on these constraints, new input data are generated that cause a NULL pointer dereference and make the program crash. Table 9 presents the result of testing the second scenario.

Although our implemented algorithm detected the vulnerabilities in the test programs in a short time (less than 10 s on average), the test duration would be more for real-world large applications. In large programs, there are a huge number of feasible execution paths and therefore a huge number of constraints would be generated. This problem is known as path explosion. Various techniques are proposed against path explosion in concolic execution, such as parallelization [37,38]. Since our concern is on extendibility, we have not implemented these techniques in our vulnerability seeking algorithm. We can revise the algorithm in the future based on the recent enhances in concolic execution technique to increase its performance.

7. Conclusion

In this paper, we presented a first step towards designing an extendable vulnerability detection method for the executable codes. To be extendable, the vulnerability seeking algorithm should be independent from the specified vulnerabilities. Thus, a general specification method is required that is semantically understandable by the vulnerability seeking algorithm and covers all vulnerability classes even the ones that will be discovered in the future. In this paper, vulnerabilities are modeled in a two-element structure {Container(s), Rule}. We also presented a formal definition for single-instruction vulnerabilities. We considered the detection of a specified vulnerability as solving a satisfaction problem.

Based on the proposed model, vulnerabilities are specified using the Vex language to be detected in the executable codes. The vulnerability seeking algorithm searches through program instructions for containers of vulnerabilities. When there is a match, the related rule is translated into a set of constraints and is combined with the constraints of the current execution path. These constraints are queried from STP and, if feasible, new input data are generated to detect the vulnerability.

The algorithm can be extended to detect a new vulnerability with inserting the containers of the vulnerability to the set of vulnerability containers and adding the relevant vulnerability constraint calculation routines to it. As an evidence that the proposed method is extendable, we adopted our {Container(s), Rule} model for the specification of five single-instruction vulnerabilities. The experiments demonstrated that our proposed vulnerability detection method can detect all the specified vulnerabilities in the test programs.

The general model helps in specifying vulnerabilities more systematically. Thus, we could present a more comprehensive specification for three classes of integer bugs that were more successful than the previous work in our experiments. However, extracting the containers and the rule for a specific vulnerability is still heuristic and it requires an algorithmic method for generating containers and the rules.

In the future, we are going to extend the specification method to cover more complicated vulnerability classes. According to our abstract point of view, complicated vulnerabilities are again created when improper data are assigned to specific containers. For extending the specification method to cover multi-instruction vulnerabilities, our intuition is that multi-instruction vulnerabilities appear through a sequence of instructions in the program. Therefore, they are specifiable through a sequence of pairs of {Container(s), Rule}. So the model should be enhanced so that vulnerabilities be specified through a sequence of one or more pairs of {Container(s), Rule}. Also, the vulnerability seeking algorithm should be revised to generate appropriate vulnerability constraints for the specified vulnerability rules. We have recently extended our algorithm to detect heap-based and stack-based buffer overflow vulnerabilities in executable codes and it achieved acceptable results in the experiments [28,29]. It shows the possibility of extending the proposed vulnerability seeking algorithm to detect multi-instruction vulnerabilities. However, we postponed formal specification of these vulnerabilities to our future works.

One challenge in using the concolic execution method in our vulnerability seeking algorithm is path explosion. In fact, the num-

ber of feasible execution paths increases exponentially when applying concolic execution to test large programs. Various optimization techniques have been proposed against the path explosion [39], which are not currently implemented in our vulnerability seeking algorithm. In the future we are going to improve the performance of our algorithm by applying appropriate optimization techniques to tackle the path explosion challenge.

We are also going to define more rules for the integer vulnerabilities. Currently, we only specified the vulnerability for 32-bit integer data types. Other data types, such as short or long, should be considered in our future works.

References

[1] M. Almorsy, J. Grundy, A.S. Ibrahim, Supporting automated vulnerability analysis using formalized vulnerability signatures, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2012, pp. 100–109.

[2] J.-E.J. Tevis, J.A. Hamilton Jr, Static analysis of anomalies and security vulnerabilities in executable files, in: Proceedings of the 44th annual Southeast regional conference, ACM, 2006, pp. 560–565.

[3] N. Dor, M. Rodeh, M. Sagiv, Cssv: Towards a realistic tool for statically detecting all buffer overflows in c, in: ACM Sigplan Notices, vol. 38, ACM, 2003, pp. 155–167.

[4] V. Ganapathy, S. Jha, D. Chandler, D. Melski, D. Vitek, Buffer overrun detection using linear programming and static analysis, in: Proceedings of the 10th ACM conference on Computer and communications security, ACM, 2003, pp. 345–354.

[5] D. Wagner, J.S. Foster, E.A. Brewer, A. Aiken, A first step towards automated detection of buffer overrun vulnerabilities., in: NDSS, 2000, pp. 2000–2015.

[6] A. Sotirov, Automatic vulnerability detection using static analysis, MSc thesis, The University of Alabama, 2005 Ph.D. thesis. http://gcc.vulncheck.org/sotirov05automatic.pdf.

[7] J. Yang, T. Kremenek, Y. Xie, D. Engler, Meca: an extensible, expressive system and language for statically checking security properties, in: Proceedings of the 10th ACM conference on Computer and communications security, ACM, 2003, pp. 321–334.

[8] D. Balzarotti, M. Cova, V.V. Felmetsger, G. Vigna, Multi-module vulnerability analysis of web-based applications, in: Proceedings of the 14th ACM conference on Computer and communications security, ACM, 2007, pp. 25–35.

[9] M.S. Lam, M. Martin, B. Livshits, J. Whaley, Securing web applications with static and dynamic information flow tracking, in: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, ACM, 2008, pp. 3–12.

[10] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, M. Veanes, Fast and precise sanitizer analysis with bek, in: Proceedings of the 20th USENIX conference on Security, USENIX Association, 2011.

[11] U. Shankar, K. Talwar, J.S. Foster, D. Wagner, Detecting format string vulnerabilities with type qualifiers., in: USENIX Security Symposium, 2001, pp. 201–220.

[12] C. Cadar, D. Dunbar, D.R. Engler, Klee: unassisted and automatic generation of high-coverage tests for complex systems programs., in: OSDI, vol. 8, 2008, pp. 209–224.

[13] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, D.R. Engler, Exe: automatically generating inputs of death, ACM Trans. Inf. Syst. Secur. (TISSEC) 12 (2) (2006) 10.

[14] D. Molnar, X.C. Li, D. Wagner, Dynamic test generation to find integer bugs in x86 binary linux programs., in: USENIX Security Symposium, 2009, pp. 67–82.

[15] G. Balakrishnan, T. Reps, Wysinwyx: what you see is not what you execute, ACM Trans. Program. Lang. Syst. (TOPLAS) 32 (6) (2010) 23.

[16] MITRE-CWE, CWE-369: division by zero, 2014a, (http://cwe.mitre.org/data/definitions/369.html). [Online; accessed 2015-02-03].

[17] MITRE-CWE, CWE-190: integer overflow or wraparound, 2014b, (http://cwe.mitre.org/data/definitions/190.html). [Online; accessed 2015-02-03].

[18] MITRE-CWE, CWE-191: integer underflow (Wrap or Wraparound, 2014c, (http://cwe.mitre.org/data/definitions/190.html).[Online; accessed 2015-02-03].

[19] MITRE-CWE, Cwe-681: incorrect conversion between numeric types, 2014d, (http://cwe.mitre.org/data/definitions/681.html). [Online; accessed 2015-02-03].

[20] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, in: ACM Sigplan Notices, vol. 42, ACM, 2007, pp. 89–100.

[21] MITRE-CWE, CWE-476: null pointer dereference, 2014, (http://cwe.mitre.org/data/definitions/476.html). [Online; accessed 2015-02-03].

[22] D. Byers, S. Ardi, N. Shahmehri, C. Duma, Modeling software vulnerabilities with vulnerability cause graphs, in: Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on, IEEE, 2006, pp. 411–422.

[23] W. Mallouli, A. Mammar, A. Cavalli, W. Jimenez, Vdc-based dynamic code analysis: application to c programs, J. Internet Serv. Inf. Secur. 1 (2/3) (2011) 4–20.

[24] MITRE, Common vulnerabilities and exposures, 2014a, (https://cve.mitre.org/index.html). [Online; accessed 2015-02-03].

[25] MITRE, Common weakness enumeration, 2014b, (http://cwe.mitre.org/). [Online; accessed 2015-02-03].

[26] B. Livshits, Improving software security with precise static and runtime analysis, Stanford University, 2006 Ph.D. thesis.

[27] T. Vajk, G. Mezei, T. Levendovszky, An incremental ocl compiler for modeling environments, Electron. Commun. EASST 15 (2008).

[28] M. Mouzarani, B. Sadeghiyan, M. Zolfaghari, A smart fuzzing method for detecting heap-based buffer overflows in executable codes., in: PRDC, 2015, pp. 42–49.

[29] M. Mouzarani, B. Sadeghiyan, M. Zolfaghari, A smart fuzzing method for detecting stack-based buffer overflows in binary codes, IET Software 10 (4) (2016) 96–107.

[30] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, Dependable Secure Comput. IEEE Trans. 1 (1) (2004) 11–33.

[31] P. Godefroid, N. Klarlund, K. Sen, Dart: directed automated random testing, in: ACM Sigplan Notices, vol. 40, ACM, 2005, pp. 213–223.

[32] G. Campana, Fuzzgrind: an automatic fuzzing tool, 2009, (http://esec-lab.sogeti.com/pages/Fuzzgrind/). Accessed 10-12-2015.

[33] I. Haller, A. Slowinska, M. Neugschwandtner, H. Bos, Dowsing for overflows: a guided fuzzer to find buffer boundary violations., in: USENIX Security, 2013, pp. 49–64.

[34] P. Godefroid, M.Y. Levin, D. Molnar, Sage: whitebox fuzzing for security testing, Queue 10 (1) (2012) 20–27.

[35] V. developers, Valgrind user manual, September 2014, (http://valgrind.org/docs/manual/manual.html). [Online; accessed 2015-02-03].

[36] Q. Gu, L. Zhu, Z. Cai, Evaluation measures of the classification performance of imbalanced data sets, in: International Symposium on Intelligence Computation and Applications, Springer, 2009, pp. 461–471.

[37] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, G. Candea, Cloud9: a software testing service, ACM SIGOPS Operating Syst. Rev. 43 (4) (2010) 5–10.

[38] V. Chipounov, V. Kuznetsov, G. Candea, S2E: a platform for in-vivo multi-path analysis of software systems, vol. 47, ACM, 2012.

[39] T. Chen, X.-s. Zhang, S.-z. Guo, H.-y. Li, Y. Wu, State of the art: dynamic symbolic execution for automated test generation, Future Gener. Comput. Syst. 29 (7) (2013) 1758–1773.