2018

Building an Abstract-Syntax-Tree-Oriented Symbolic Execution Engine for PHP Programs

Jin Huang
Wright State University

Building An Abstract-Syntax-Tree-Oriented Symbolic Execution Engine for PHP Programs

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

by

Jin Huang
M.S. Material Sciences, University of Florida, 2014
B.S. Material Sciences, Beijing Technology and Business University, 2010

2018
Wright State University

Wright State University
GRADUATE SCHOOL

April 11, 2018

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Jin Huang ENTITLED Building An Abstract-Syntax-Tree-Oriented Symbolic Execution Engine for PHP Programs BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science.

Junjie Zhang, Ph.D.
thesis Director

Mateen Rizki, Ph.D.
Chair, Department of Computer
Science and Engineering

Committee on
Final Examination

Junjie Zhang, Ph.D.

Krishnaprasad Thirunarayan, Ph.D.

Phu H. Phung, Ph.D.

Barry Milligan, Ph.D.
Interim Dean of the Graduate School

ABSTRACT

Huang, Jin. M.S., Department of Computer Science and Engineering, Wright State University, 2018. *Building An Abstract-Syntax-Tree-Oriented Symbolic Execution Engine for PHP Programs.*

This thesis presents the design, implementation, and evaluation of an abstract-syntax-tree-oriented symbolic execution engine for the PHP programming language. As a symbolic execution engine, our system emulate the execution of a PHP program by assuming that all inputs are with symbolic rather than concrete values. While our system inherits the basic definition of symbolic execution, it fundamentally differs from existing symbolic execution implementations that mainly leverage intermediate representation (IRs) to operate. Specifically, our system directly takes the abstract syntax tree (AST) of a program as input and subsequently interprets this AST. Performing symbolic execution using AST offers unique advantages. First, it enables one-to-one mapping between the source code and the analysis results such as control flows and data flows. Second, it makes possible the direct instrumentation on source code to enable developer-aware changes. Third, it has higher applicability since IR is not always available. The design and implementation of our symbolic execution engine essentially feature an interpreter that interprets the AST based on symbolic values. Different from an interpreter that deterministically follows a single execution path by operating on concrete input values, the interpreter we have built needs to generate all paths, where each path has a constraint and its own environment. Constraints and environments of paths need to be dynamically created and maintained while the AST is evaluated. Our interpreter is context-dependent, where all user-defined functions are faithfully when they are called. Once all paths for a program is generated, we will automatically translate the constraint of each path into assertions that can be verified by satisfiability modulo theories (SMT) solver (e.g., Z3). The SMT solver can further verifies assertions for each path and report i) concrete input values that enable this path or ii) the infeasibility of this path. We have tested our system using both prototype PHP programs

and real PHP programs collected from WordPress plugins. The experimental results have demonstrated our system is highly effective in performing symbolic execution.

Contents

List of Figures

List of Tables

Listings

Acknowledgment

I would like to take this opportunity to appreciate my advisor, Dr. Junjie Zhang, for his guidance and mentoring towards the completion of this thesis. His expertise has been invaluable and he has tremendously contributed to my education at Wright State. I would also like to thank Dr. Krishnaprasad Thirunarayan and Dr. Phu H. Phung for volunteering their time to serve on my thesis committee and sincerely appreciate their input and expertise in evaluating this work. I would like to extend my thanks to everyone who has invested to assist me during my graduate study at Wright State University. I would like to thank my family for their support during my academic career. Finally, I would like to thank my girlfriend for supporting me while I pursued further education and for her belief in me.

Dedicated to

My Father, Mr. Zhenying Huang,

My Mother, Mrs. Fengying Zhang, and

My Girlfriend, Miss. Zhe Liu

Abbreviations

AST — Abstract Syntax Tree

SMT — Satisfiability Modulo Theories

IR — Intermediate Representation

OOP — Object Oriented Programming

PHP — PHP Hypertext Preprocessor

Introduction

Symbolic execution [1] is a fundamental method that is widely used in various areas such as software testing [2], reverse engineering [3], and vulnerability discovery [4]. The basic idea is to assign certain program variables with symbolic values and execute a program symbolically [5]. A few symbolic execution engines have been implemented, where salient examples include KLEE [6], Java PathFinder (a.k.a JPF) [7], S2E [8] and angr [9]. All these symbolic execution engines, however, take as inputs either intermediate representations (IRs) or binaries [10, 11, 12, 13, 14]. Since neither IR nor binaries offer direct mapping to the source code, it is challenging to directly map symbolic execution results to source code, offering limited information to developers and analysts. In addition, any mitigation solutions, such as patching or vulnerability fixing, will be enforced at the IR or binary level, staying transparent to developers. Last but not the least, tools and libraries for IR and binary generation are not pervasively available for all programming languages, drastically limiting the applicability of existing symbolic execution systems. In order to overcome these challenges, we have designed and implemented a symbolic execution engine for PHP programs with the following objectives:

- Source-Code-Driven Symbolic Execution: The analysis results can enable one-to-one mapping between the analysis results and source code. For example, our symbolic engine can automatically discover all statements that impact the execution of a path by integrating data flow analysis and control flow analysis.

- Context-Aware: The symbolic execution for user-defined functions will be context aware. Specifically, we will perform symbolic execution for each function when it is called, depending on its context information.

Building a symbolic execution engine with these objectives, however, is faced with significant challenges. First, the source code of a program usually presents much more variety compared to IR or binary instructions (e.g., X86). It features huge syntactical diversity such as nested branches and loops. Comparatively, IRs and binaries are usually characterized by a small set of instructions and registers. Second, IRs, such as LLVM [15], usually support single static assignments (SSA) [16], which makes the maintenance of variables straightforward. Source code, instead, usually has intensive reuse of variables. Third, we need to dynamically generate and maintain constraints together with variables for each path, which imply significant implementation challenges. In order to systematically overcome these challenges, we have make the following contributions:

- Designing an interpreter to perform symbolic execution based on abstract syntax trees [17] (AST) of a program: Our system interprets the AST of a program using inputs with symbolic values. For each possible execution path, our program maintains an environment and a constraint. An environment contains all variables and their symbolic values or symbolically-derived values; a constraint is the condition to be satisfied to execute this path. Once a branch statement (e.g., the "if", "switch", or "for" statement) is evaluated, new paths are created and their corresponding environments are generated.

- Designing an interpreter to convert path constraint into Satisfiability Modulo Theories (SMT) expressions: For each path constraint, our system will interpret it and automatically generate constraints based on boolean, integer, and/or string symbolic values, which are verified by existing SMT solvers [18].

Our current implementation focuses on PHP, one of the most popular programming

languages for developing backend web services. PHP is an object oriented programming languages with dynamic type. Our symbolic execution engine considers all core language features including **super global variables**, **unary operations**, **binary operations**, **function definitions**, **function calls**, **assignment statements**, and **control statements**. We use hash table to implement the environment to accomplish efficient search. A key of the hash table is corresponding to the name of a variable and its value is a reference to an object, where this object represents a symbolic value or a derived symbolic value. We have used a tree data structure, which actually implements S-expression [19], to represent the constraint for each path. Each variable inside the path constraint is a reference to an object, where this object represents a symbolic value or a derived symbolic value. Our system also translates the tree-based path constraint into the constraint in the format of Z3 [20], a SMT solver. We then leverage Z3 to verify each path constraint to evaluate its feasibility.

In order to evaluate our system, we have collected $1,377$ plugins from the WordPress plugin repository [21], where WordPress is the most popular open-source content-sharing platform. These plugins include a varity of language features and amass a large number of code. Experimental results have demonstrated that our system can effectively interpret all these plugins and generate path constraints.

The remaining of this thesis is organized as follows. Chapter 3 presents the design of the system and implementation. Experimental results are presented in Chapter 4 and Chapter 5 concludes the thesis.

Related Work

Nguyen et al. provided a tool (Varis) for supporting the additional editor services on the client-side code which was dynamically generated in a PHP-based web application. This tool was provided in the integrated development environments (IDEs) for dynamic web applications. Although the traditional IDEs have provided a complete editor services for traditional software applications, supporting the dynamic client-side code generated in web application is difficult. Because the client-side code, wrote in client-side languages such as HTML, is dynamically generated from the server-side code, wrote in service-side languages such as PHP, and is embedded as string type literals in the server-side PHP program. Existing tool to the date before the authors work provide either server-side code or the generated client-code, but do not support the editor services for the dynamically generated client-code. The symbolic execution was used to estimate all possible dynamically generated client-code and parse the ASTs of the dynamic client-code in the VarDOM, a sub-component in Vairs. Following this, the VarDOM was support all variables, expressions, and statements about the dynamic client-code in the server-side program. Vair could support various types of editor services in IDE, such as syntax highlighting, code completion, and other types of code analysis.

Ehresmann et al. developed a PHP Analysis and Regression Testing Engine (PARTE), a tool which could effective take a regression testing for the frequently patched or revision PHP web applications. Rather than applying regression testing to the entire program, the author instead utilized PARTE to identify the affected code areas for the two consecutive

versions code changing using impact analysis. To perform impact analysis, The PARTE use the High Intermediate Representation (HIR) and Abstract Syntax Trees (ASTs) to construct program dependence graphs (PDGs) of two consecutive versions of PHP-based program. And then, identify the difference code areas in two consecutive version codes. To test the updated feature or new functionalities in the affected areas of program, a new test case generation method was designed by the author that generates new executable test cases by using both string type and numeric type input value in the program slices. Based on these, the PARTE can effectively decrease the necessary general test cases and focus only on the impact areas in upgraded frequently web applications.

Son and Shmatikov developed static security analysis tool for PHP applications (SAFER-PHP). The SAFERPHP is the first semantic security analysis tool to detect the infinite loop trigger bugs and missing authorization vulnerability, and also the first security analysis tool to support the objected-oriented features of PHP based web program. The standard tainted analysis, the algorithm based on symbolic execution, and the algorithm based on inter-procedural algorithms were utilized in the semantic security analysis. The SAFERPHP parse the PHP source code and generate the ASTs of the code. After that, the SAFERPHP build the call graph and the control-flow graph in whole-program. The critical variables, which will execute the sensitive operations, were collected from the program call graph and the control-flow graph. A loop whose termination decide by the external inputs will be find by the taint analysis, and the symbolic execution is to check the infinite loop caused by the program. Based on the SAFERPHP two classes of vulnerabilities, i) denial-of-service, ii) authorization missing check, could be detected toward this semantic analysis. The SAFER-PHP detect unreported vulnerabilities from the open-source PHP applications.

Design and Implementation

Our symbolic engine is mainly composed of two phases. In the first phase, it takes as input the AST of a PHP program and generate path constraints. In the second phase, it translates each path constraint into Z3 constraint for automated verification. Figure 3.1 presents the architectural overview of the proposed system. In this chapter, we will first introduce the core data structure for environments and path constraints. We then discuss the interpretation and finally present the translation of path constraints into Z3 constraints.

3.1 Core Data Structures

Since the both the interpretation and the translation rely on core data structures, we will first introduce our core data structures. The proposed symbolic execution engine relies on two critical data structures including i) the environment and ii) the path constraint. The environment needs to be frequently accessed to create, retrieve, and update variables and their values. Path constraints need to support incremental expansion as new constraints in different formats (e.g., AND, OR, and NEGATE) will be added as the program is interpreted.

We therefore adopt hash table to implement the environment to support efficient random access. For each $< key, value >$ pair in the hash table, the key is the name of a variable and the value refers to its variable value. It is worth noting that the variable could

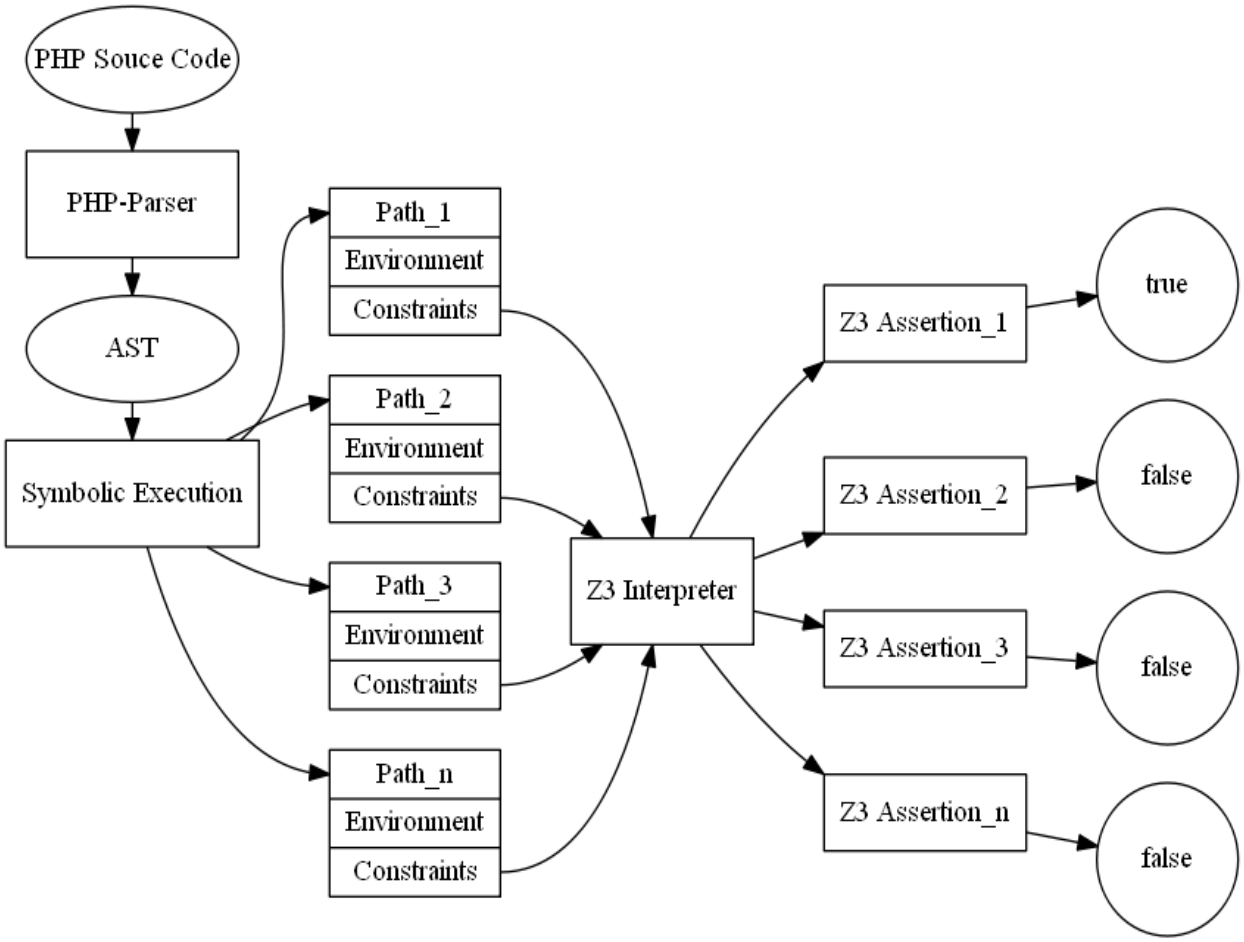

Figure 3.1: System Overview

be a concrete value, a symbolic value, or a derived value from either or both of them.

We have used a tree data structure, which actually implements S-expression, to represent the constraint for each path. A leaf node of this tree represents either a concrete value or a symbolic value. A non-leaf node represents an operation node (e.g., either a unary operation or a binary operation).

The Listing 3.1 presents a PHP program, which has totally 3 paths. Since the variable $a and $b get inputs from global variables that are not known in advance, their values are initialized as symbolic values. The Listing 3.2 presents environments (i.e., hash tables) for three paths (on the completion of each path), respectively. Figure 3.2 present path constraints for 3 paths, respectively.

Listing 3.1: PHP Codes Example

```
$a = $_POST['query'];
$b = $_REQUEST['action'];
$c = 10;

$output = '';

if ( $c < 0 ) {
    $e      = true;
    $output = "Success";

} elseif ( $c < 99 && $c > 0 ) {
    $e      = false;
    $output = 'Success';

} else {
    $output = 'Failure';
}
```

Listing 3.2: Global Environment

```
Final paths:

Path index: 1
Environment:
a => (_POST_query_symbol:symbol_SuperGlobal)
b => (_REQUEST_action_symbol:symbol_SuperGlobal)
c => 10:int
output => Success:string
e => true:bool


Path index: 2
Environment:
a => (_POST_query_symbol:symbol_SuperGlobal)
b => (_REQUEST_action_symbol:symbol_SuperGlobal)
c => 10:int
output => Success:string
e => false:bool

Path index: 3
Environment:
a => (_POST_query_symbol:symbol_SuperGlobal)
b => (_REQUEST_action_symbol:symbol_SuperGlobal)
c => 10:int
output => Failure:string
```

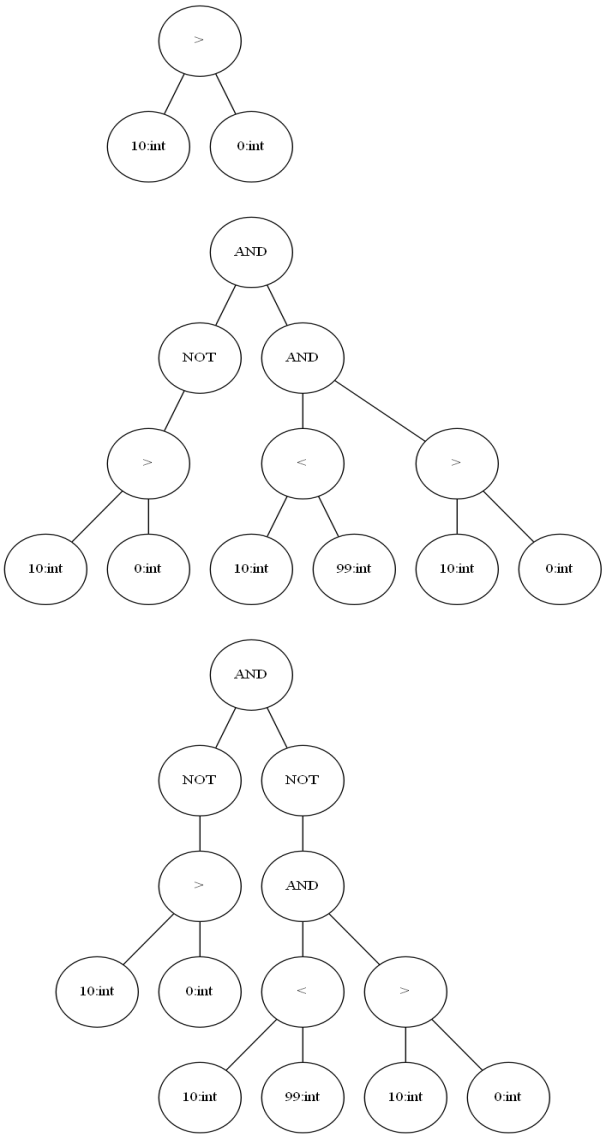

Figure 3.2: Path Constraints

3.2 Interpretation

Our symbolic execution engine will recursively interpret the AST of a PHP program to generate path constraints by interacting with environments. We currently interpret core language features including **super global variables**, **unary operations**, **binary operations**, **function definitions**, **function calls**, **assignment statements**, **control statements**. We will discuss each of them in the following section. In addition, we will specifically illustrate how our system handles access to single- or multi-dimensional arrays.

3.2.1 Superglobal Variables and Uninitialized Variables

There are two types of variables whose values will be assigned with symbolic values, namely the superglobal variables and uninitialized variables.

- Superglobal variables are built-in variables for PHP. A superglobal variable contains the input from the external user of the studied program, which is unpredictable. Therefore, we set its value as a symbolic value. We currently consider all superglobal variables in PHP including "_POST", "_GET", "_COOKIE", "_REQUEST", "_SERVER", and "_SESSION".

- Uninitialized Variables could be observed in a PHP program when a complete over view of the entire program is unavailable.

The Listing 3.3 presents two examples for superglobal variables and uninitialized variables, respectively. The Listing 3.4 presents its AST-based representation.

Listing 3.3: Symbol Value Example

```
1 $varFromSuperGlobal = $_POST['filename'];
2
3 $varFromExternalInput = $var1;
```

Listing 3.4: AST for Symbol Value

```
1  0: Expr_Assign(
2      var: Expr_Variable(
3          name: varFromExternalInput
4      )
5      expr: Expr_Variable(
6          name: varable
7      )
8  )
9  1: Expr_Assign(
10     var: Expr_Variable(
11         name: varFromSuperGlobal
12     )
13     expr: Expr_ArrayDimFetch(
14         var: Expr_Variable(
```

```
            name: _POST
        )
        dim: Scalar_String(
            value: filename
        )
    )
)
```

Specifically, "$_POST['filename']" refers to the value offered by an external user through the POST method, therefore remaining unknown. As a result, "$varFromSuperGlobal" will be assigned with a symbolic value. The Listing 3.5 presents the evaluation of AST node of superglobal variables to generate symbolic values.

The $var1 in Listing 3.3 is assumed to be an uninitialized variable in this specific example. Therefore, when interpreting $var1, our system will automatically assign a symbolic value to $var1. The evaluation of the assignment operation, which will be discussed later, will assign $var1's value, which is currently a symbolic value, to the variable $varFromExternalInput. The Listing 3.6 presents how our system assigns symbolic values to uninitialized variables. Specifically, when we cannot find a variable that is defined in the environment of a path, we add this variable into the environment and assign a symbolic value to this variable.

Listing 3.5: Superglobal Variable Evaluation

```
if ( $arrayName == "_POST"    || $arrayName == "_GET" ||
     $arrayName == "_COOKIE" || $arrayName == "_REQUEST" ||
     $arrayName == "_SERVER" || $arrayName == "_SESSION" )  {

        $a = new leafNodeSymbol_SG( $arrayName . "_" .
                                    $arrayIndex->getValue() .
                                    "_symbol" );

            return $a;
    }
```

Listing 3.6: Variable Expression Evaluation

```
case "PhpParser\Node\Expr\Variable":

```

```
3     $resultDict = $env->getVariable( $node->name );
4
5     if ( empty( $resultDict ) ) {
6        $resultDict   = array();
7        $envPathIndex = $env->get_path_index();
8
9        foreach ( $envPathIndex as $index ) {
10          $temp = new Symbol_String( "_Unknow_Argument_" );
11
12          $resultDict[ $index ] = $temp;
13       }
14
15       return $resultDict;
16    }
17
18    return $resultDict;
```

Listing 3.7 presents the resulted environment. Since there is only one path, only one environment is resulted. All these variables are associated with symbolic values. The "var1" is an uninitialized variable; the "varFromExternalInput" is assigned with the value of "var1"; the "varFromSuperGlobal" derives its symbolic value from the superglobal variable.

Listing 3.7: Global Environment

```
1 path idx: 1
2
3 Condition: None.
4 Environment:
5 var1 => (_Unknow_Argument_:symbol_String)
6 varFromExternalInput => (_Unknow_Argument_:symbol_String)
7 varFromSuperGlobal => (_POST_filename_symbol:symbol_SuperGlobal)
```

3.2.2 Unary and Binary Expressions

The unary and binary expressions are mainly used in branch/control statements. An unary operation is represented as "u-op e", where "u-op" is an unary operation and "e" is the expression. A binary operation is represented as "e1 bin-op e2", where "bin-op" is a binary operation and "e1" and "e2" are two expressions. We will focus on discussing binary expressions. Generally, we will evaluate both "e1" and "e2". It will then combine results

of these two expressions using the “bin-op”. Listing 3.8 presents our design for the “equal” binary expression. It first evaluates the left expression and next the right expression. Then it creates a node that combines both left result and right result using the “equal” operation. It is worth noting that each path has its own value for an expression. Therefore, each path will has its own node based on the “equal” operation.

Listing 3.8: Equal Expression Evaluation

```
case "PhpParser\Node\Expr\BinaryOp\Equal":
        $valueDictLeft  = eval_node( $node->left, $env );
        $valueDictRight = eval_node( $node->right, $env );
        $resultDict     = expEqual( $valueDictLeft, ...
            $valueDictRight );

        return $resultDict;
...

function expEqual( $valueDict1, $valueDict2 ) {

    $result = array();

    $ks = array_keys( $valueDict1 );
    foreach ( $ks as $k ) {
        $value        = new opEqual( $valueDict1[ $k ], ...
            $valueDict2[ $k ] );
        $result[ $k ] = $value;
    }

    return $result;
}
```

Listing 3.9 shows a PHP code example and Listing 3.10 illustrates the evaluation result. In this example, the binary operation “Equal” was utilized in the test condition clause. And the “Equal Expression” was added in the condition constraint in then environment.

Listing 3.9: Equal Expression Example

```
$bool = true;

if ($bool == true){
   $output = true;
} else {
   $output = false;
}
```

Listing 3.10: The Result of Example

```
Path index: 1
Condition: (operator: Equal true:bool,true:bool,)

Environment:
bool => true:bool
output => true:bool



Path index: 2
Condition: (operator: NOT (operator: Equal true:bool,true:bool,),)

Environment:
bool => true:bool
outputt => false:bool
```

3.2.3 Assignment

An assignment statement is expressed in the format of “v = e”, where “v” is a variable and “e” is an expression. Our system will evaluate “e” for each path and assign the result to “v” for each path. Listing 3.11 presents our system to evaluate the assignment. Specifically, it evaluate “e” for all paths and then assign values back to “v” in environments corresponding to their paths.

Listing 3.11: Assignment Statement Evaluation

```
case "PhpParser\Node\Expr\Assign":

    $var  = $node->var;

    $expr = $node->expr;

    $var_name = $var->name;

    $expResult = eval_node( $expr, $env );

    $env->setVariable( $var_name, $expResult );

    return;
```

Listing 3.12 presents a sequence of assignment statements and Listing 3.13 demonstrates it AST-based representation. Specifically, a variable was assigned two times with different variable. In this example, the variable $a was assigned by number 5 and $b was assigned by number 9 when evaluated the third statement$c = $a + $b. So variable $c is assigned by 14:int. After execution forward, the variable $b was assigned by number 100 so the $b, in the environment, was overwritten by integer 100. When the variable was reused in the assignment statement: $d = $a + $b , the variable $d was assigned by 105:int.

Listing 3.12: Assignment Statements Example

```
$a = 5;
$b = 9;
$c = $a + $b;

$b = 100;
$d = $a + $b;
```

Listing 3.13: AST for Assignment Statement

```
array(
    0: Expr_Assign(
        var: Expr_Variable(
            name: a
        )
        expr: Scalar_LNumber(
            value: 5
        )
    )
    1: Expr_Assign(
        var: Expr_Variable(
            name: b
        )
        expr: Scalar_LNumber(
            value: 9
        )
    )
    2: Expr_Assign(
        var: Expr_Variable(
            name: c
        )
        expr: Expr_BinaryOp_Plus(
```

```
            left: Expr_Variable(
                name: a
            )
            right: Expr_Variable(
                name: b
            )
        )
    )
    3: Expr_Assign(
        var: Expr_Variable(
            name: b
        )
        expr: Scalar_LNumber(
            value: 100
        )
    )
    4: Expr_Assign(
        var: Expr_Variable(
            name: d
        )
        expr: Expr_BinaryOp_Plus(
            left: Expr_Variable(
                name: a
            )
            right: Expr_Variable(
                name: b
            )
        )
    )
)
```

Listing 3.14: Assignment Example Results Example

```
Path index: 1
Condition: None.

Environment:
a => 5:int
b => 100:int
c => (operator: plus 5:int,9:int,)
d => (operator: plus 5:int,100:int,)
```

3.2.4 Function Call

The evaluation of a function call features the i) design of a local environment and ii) the return of evaluated result. In order to demonstrate our design, we will start with an

example. Listing 3.15 and Listing 3.16 show a session of PHP example and the AST for the function call in this example, respectively.

Listing 3.15: Example for Function Implementation

```php
function positive_sum(int $x, int $y){
   $localSum = 0;

   if ($x >0 && $y > 0){
      $localSum = $x + $y;
   } else {
      $localSum = -1;
   }

   return $localSum;
}


$a = 5;
$b = 9;
$c = 0;

if ( $a < 0 ) {
   $a = 10;
   $b = 99;
} elseif ( $a > 10 ) {
   $b = $a;
} else {
   $c = 99999;
}

$sum = positive_sum($a, $b);
```

Listing 3.16: AST for for Function Implementation

```
expr: Expr_FuncCall(
    name: Name(
        parts: array(
            0: positive_sum
        )
    )
    args: array(
        0: Arg(
            value: Expr_Variable(
                name: a
            )
            byRef: false
            unpack: false
        )
```

```
15          1: Arg(
16              value: Expr_Variable(
17                  name: b
18              )
19              byRef: false
20              unpack: false
21          )
22      )
23  )
```

There are two components in the function call expression. One is the function name that was referenced by "name" in AST. Another one is the passing arguments referenced by "args". The number of passing arguments may be either zero or more than one. Each argument has different values in different local environments. Therefore, when an argument is passed into a function call, we pass a dictionary of all values of this argument in local environments. The implementation for supporting this operation is shown in Listing 3.17. The variable $index is the index of each entry in dictionary $dictParameterIndexAndValues. The value of each entry contains a dictionary, where the key is the index of each local path and the value is the value of this variable corresponding to this path.

Listing 3.17: Function Call Evaluation

```
1  case "PhpParser\Node\Expr\FuncCall":
2     $func_name = $node->name->parts[0];
3
4     $args = $node->args;
5
6     $dictParameterIndexAndValues = array();
7
8     $index = 0;
9
10    foreach ( $args as $v ) {
11       $a = eval_node( $v->value, $env, $layer + 1 );
12       $dictParameterIndexAndValues[ $index ] = $a;
13       $index ++;
14    }
```

After processing the argument of function call, we need to check whether the implementation of the function is an internal function in PHP language or a user-declared function. If it is an internal function, we model the internal function as a symbol value

returned as the result of this function. The implementation for supporting internal function call evaluation is shown in List 3.18.

Listing 3.18: Function Call Evaluation

```
case "PhpParser\Node\Expr\FuncCall":

    $result_from_built_in = model_built_in_functions( $func_name,
        $dictParameterIndexAndValues, $env );


    if ( $result_from_built_in != null ) {
        return $result_from_built_in;
    }
```

If it is a user-declared function, then this function could be fetched from the User Declaration Function Buffer (UDFBuffer), which is a dictionary structure for storing all AST of the user-declare function that was parsed by the AST interpreter during the source code evaluation. In the User Declaration Function Buffer, each AST of function was reference by the its own name, so the user-declared functions could be fetched by their name to further evaluation. In this work, we create a local environment for the function evaluation. The local environment was initialized by the input arguments that have been processed above. After the local environment standing by, we evaluated all statements of function under this new local environment. The implementation for supporting user declared function call evaluation is shown in Listing 3.19.

Listing 3.19: Function Call Evaluation

```
case "PhpParser\Node\Expr\FuncCall":

    global $funcArray;
    if ( ! array_key_exists( $func_name, $funcArray ) ) {
        echo "ERROR: function " . $func_name . " is called before ...
            defined!\n";
        die();
    }

    $func = $funcArray[ $func_name ];

    $dictParameterIndex_And_Name_And_DefaultValue =
```

```
12        $func->getParameterIndex_And_Name_And_DefaultValue();
13
14    $localEnv = new allPathConditionEnv();
15
16    if ( empty( $dictParameterIndex_And_Name_And_DefaultValue ) ) {
17        $path_index = $env->get_path_index();
18        $localEnv->createLocalEnvWithoutParameter( $path_index );
19    } else {
20        $localEnv->createLocalEnv( $dictParameterIndexAndValues,
21            $dictParameterIndex_And_Name_And_DefaultValue );
22    }
23
24    foreach ( $func->stmts as $stmt ) {
25        eval_node( $stmt, $localEnv, $layer + 1 );
26    }
```

Handling the function return is challenging in the implementation of the function. For each local environment, we set their parents path index during the process of local environment initialization. Therefore, we need to extract three components for each local environment: 1) it parents path index, 2) its local condition constraints that may generated during the statements evaluation, and 3) the RETURN value. After that, we join the parent index with path index in each path in global environment to expand the original path in global environment. And we "AND" the condition with the new condition, generated in the local environment in the to-be-joined local path. Finally, we add the RETURN value to each path. The implementation for supporting function call return is shown in Listing 3.20.

Listing 3.20: Function Call Evaluation

```
1  case "PhpParser\Node\Expr\FuncCall":
2
3      $funcResult =
4          $localEnv->extract_ParentIndex_Condition_ReturnValue_from_LocalEnv();
5
6      $resultOfFuncCall =
7          $env->join_with_returned_value_from_function( $funcResult, ...
               $currentLine );
8
9      return $resultOfFuncCall;
```

The Listing 3.21 shows the symbolic execution result of the example in Listing 3.15.

Listing 3.21: Function Call Evaluation

```
Path index: 1
Condition: (operator: AND (operator: < 5:int,0:int,),(operator: ...
    AND (operator: > 10:int,0:int,),(operator: > 99:int,0:int,),),)
Environment:
a => 10:int
b => 99:int
c => 0:int
sum => 109:int


Path index: 2
Condition: (operator: AND (operator: < 5:int,0:int,),(operator: ...
    NOT (operator: AND (operator: > 10:int,0:int,),(operator: > ...
    99:int,0:int,),),),)
Environment:
a => 10:int
b => 99:int
c => 0:int
sum => -1:int


Path index: 3
Condition: (operator: AND (operator: AND (operator: NOT ...
    (operator: < 5:int,0:int,),),(operator: > ...
    5:int,10:int,),),(operator: AND (operator: > ...
    5:int,0:int,),(operator: > 5:int,0:int,),),)
Environment:
a => 5:int
b => 5:int
c => 0:int
sum => 10:int


Path index: 4
Condition: (operator: AND (operator: AND (operator: NOT ...
    (operator: < 5:int,0:int,),),(operator: > ...
    5:int,10:int,),),(operator: NOT (operator: AND (operator: > ...
    5:int,0:int,),(operator: > 5:int,0:int,),),),)
Environment:
a => 5:int
b => 5:int
c => 0:int
sum => -1:int


Path index: 5
Condition: (operator: AND (operator: AND (operator: NOT ...
    (operator: < 5:int,0:int,),),(operator: NOT (operator: > ...
    5:int,10:int,),),),(operator: AND (operator: > ...
    5:int,0:int,),(operator: > 9:int,0:int,),),)
Environment:
a => 5:int
```

```
41 b => 9:int
42 c => 99999:int
43 sum => 14:int
44
45
46 Path index: 6
47 Condition: (operator: AND (operator: AND (operator: NOT ...
       (operator: < 5:int,0:int,),),(operator: NOT (operator: > ...
       5:int,10:int,),),),(operator: NOT (operator: AND (operator: > ...
       5:int,0:int,),(operator: > 9:int,0:int,),),),)
48 Environment:
49 a => 5:int
50 b => 9:int
51 c => 99999:int
52 sum => -1:int
```

3.2.5 Control Statement

Control statements lead to the creation of paths and path constraints. Our system processes Control Statements[1] in PHP code. The control statements include if statement, loop, and switch statement. The if statement and the switch statement will exponentially increase the program path. In this work, we initialize the conditional environment with one default path and declared the Environment as Null. The condition labeled with "None". The "None" means no conditional constraints.

For evaluating the if statement, we preserve the previous global environment before evaluating the if statement and expand the global environment after evaluating each component of the if statement. Figure 3.22 shows the general structure for the if control statement. There are several components under it, which include a test condition clause, some following statements, and may or may not include "elseif" or "else" components.

Listing 3.22: If Statement Rule

```
1 if (test_condition) {
2     some_statements
3 }
4 // or
5 if (test_condition) {
```

[1] http://php.net/manual/en/language.control-structures.php

```
    some_statements
} elseif (test_condition) {
    some_statements
} else {
    some_statements
}
```

First, we create a new local environment copied from the global environment. And then, it requires one buffer, called Condition Buffer, to save the result from the test condition clause evaluation. The Condition Buffer will be used when the "elseif" or "else" components are included in the if statement.

After evaluating the test condition clause, we backup the result into Condition Buffer and add the condition constraint in the local environment using AND operation. And then, we evaluate the following statements under if statement and update the statement result into its local environment. And now, we need the second buffer to save all possible local environments that were generated from the elseif or else components evaluation. This new buffer called Environment Buffer. After all components in if statement were evaluated, this system will replace the previous global environment by all the local environments kept in the Environment Buffer for forward execution.

Listing 3.23: If Statement Codes

```
<?php

$a = 5;
$b = 6;
$c = $_REQUEST['file'];

if ( $a < 10 ) {
    $b = 10;
}

if ( $a > 4 ) {
    $a = 100;
    $e = 200;
} elseif ( $a > 100 ) {
    $a = 666;
} else {
    $d = "LLLLLL";
}
```

As an example, in Listing 3.23 above shows a small section of PHP code that involved the if statements. In this session codes, two control statements are utilizing. One is the if statement, another is the if-elseif-else statement. They will expand the original path into multiple paths in the global environment, 6 paths in this example. Listing 3.24 below show the global environment that is generated from this code.

Listing 3.24: Global Evnironment

```
Path index: 1

Condition: (operator: AND (operator: < 5:int,10:int,),(operator: ...
    > 5:int,4:int,),)
Environment:
a => 100:int
b => 10:int
c => (_REQUEST_file_symbol:symbol_SuperGlobal)
e => 200:int



Path index: 2

Condition: (operator: AND (operator: NOT (operator: < ...
    5:int,10:int,),),(operator: > 5:int,4:int,),)
Environment:
a => 100:int
b => 6:int
c => (_REQUEST_file_symbol:symbol_SuperGlobal)
e => 200:int


Path index: 3

Condition: (operator: AND (operator: AND (operator: < ...
    5:int,10:int,),(operator: NOT (operator: > ...
    5:int,4:int,),),),(operator: > 5:int,100:int,),)
Environment:
a => 666:int
b => 10:int
c => (_REQUEST_file_symbol:symbol_SuperGlobal)



Path index: 4

Condition: (operator: AND (operator: AND (operator: NOT ...
    (operator: < 5:int,10:int,),),(operator: NOT (operator: > ...
    5:int,4:int,),),),(operator: > 5:int,100:int,),)
```

```
Environment:
a => 666:int
b => 6:int
c => (_REQUEST_file_symbol:symbol_SuperGlobal)



Path index: 5

Condition: (operator: AND (operator: AND (operator: < ...
    5:int,10:int,),(operator: NOT (operator: > ...
    5:int,4:int,),),),(operator: NOT (operator: > 5:int,100:int,),),)
Environment:
a => 5:int
b => 10:int
c => (_REQUEST_file_symbol:symbol_SuperGlobal)
d => LLLLLL:string



Path index: 6

Condition: (operator: AND (operator: AND (operator: NOT ...
    (operator: < 5:int,10:int,),),(operator: NOT (operator: > ...
    5:int,4:int,),),),(operator: NOT (operator: > 5:int,100:int,),),)
Environment:
a => 5:int
b => 6:int
c => (_REQUEST_file_symbol:symbol_SuperGlobal)
d => LLLLLL:string
```

For evaluating the switch statement, it is somewhat similar to the if-elseif-else statement but there are several different points needed to be concerned. The Listing 3.25 show the structure of switch statement. The components of switch statement include one text expression, multiple case blocks, which blocks has an expression constant and some following statements, and a default case block.

Listing 3.25: Switch statement rule

```
switch ( text_expression ){
   case expression_constant_1:
      statements;
      break;
   case expression_constant_2:
      statements;
      break;
```

```
    ...
    case expression_constant_N:
       statements;
       break;
    default:
       statements;
       break;
}
```

The switch statement is similar to the if-elseif-else statement that with multiple elseif components. For each case block, the expression_constant is compared with the test expression. We use the equal operation to represent these two expressions match. We label this equal constraint to its local environment and save it into the Condition Buffer. After that, we evaluate the following statements under local environment and save the local environment into the Environment Buffer. Repeat the same process for each case block until the default case. The default case is a special one. Its conditional constraint is negated all previous cases conditional constraints. After adding all local environment into Environment Buffer and evaluating all statements, the system will replace the previous global environment by all the local environments for forward execution.

Listing 3.26: Switch Statement Codes

```
<?php

$str = 'abc';
$status= '';

switch ( $str ) {
    case 'abc':
        $status = 'case 1 matched!';
        break;

    case 'xyz':
        $status = 'case 2 matched!';
        break;

    default:
        $status = 'No case matched!';
        break;
}

```

```
20 echo $status;
```

As an example, the listing 3.26 above shows a small section of PHP code that including switch statements. In this session codes, the switch statement includes two case blocks and one default block. They will expand the original path into three paths after the switch statement evaluation. The listing 3.27 below show symbolic execution result that is generated from this code.

Listing 3.27: Global Environment

```
1  Path index: 1
2
3  Condition: (operator: Equal abc:string,abc:string,)
4  Environment:
5  str => abc:string
6  status => case 1 matched!:string
7
8
9
10 Path index: 2
11
12 Condition: (operator: AND (operator: NOT (operator: Equal ...
       abc:string,abc:string,),),(operator: Equal ...
       abc:string,xyz:string,),)
13 Environment:
14 str => abc:string
15 status => case 2 matched!:string
16
17
18
19 Path index: 3
20
21 Condition: (operator: AND (operator: NOT (operator: Equal ...
       abc:string,abc:string,),),(operator: NOT (operator: Equal ...
       abc:string,xyz:string,),),)
22 Environment:
23 str => abc:string
24 statue => No case matched!:string
```

The loop statement, the Listing 3.28 show below, is the fundamental difficult in the symbolic execution. For heuristic solving the loop statement, we skip the condition statement evaluation and just evaluate one time for their statements and update all paths in the global environment. For some unknow variables, we assume them as symbol during

the evaluation. These unknow variables was declared in initialization statement in the for loops.

Listing 3.28: Loop Rule

```
while ( condition ) {
   statements;
}

//******************************

do {
   statements;
} while (condition);

//******************

for (initialization; condition; update){
   statements;
}
```

3.2.6 Array Fetch

Array accessing is challenging in AST evaluation. In this work, we only discuss one and two-dimensional array fetch. As an example, the Listing 3.29 below shows the array fetch in PHP code. In this session codes, a variable was fetched from a one-dimensional array by an integer index, and other two variables are fetched by a string index from one or two-dimensional array respectively. The Listing 3.30 below shows the AST that generated from code in the Listing 3.29.

Listing 3.29: Array Fetch Example

```
$arr = array("abc", "xyx", "U.S.A");

$str = $arr[0];

$var = $_REQUEST['action'];

$file = $_FILES['file']['name'];
```

Listing 3.30: AST for Array Fetch

```
array(
    0: AST for array declaration was avoid.
    1: Expr_Assign(
        var: Expr_Variable(
            name: str
        )
        expr: Expr_ArrayDimFetch(
            var: Expr_Variable(
                name: arr
            )
            dim: Scalar_LNumber(
                value: 0
            )
        )
    )
    2: Expr_Assign(
        var: Expr_Variable(
            name: var
        )
        expr: Expr_ArrayDimFetch(
            var: Expr_Variable(
                name: _REQUEST
            )
            dim: Scalar_String(
                value: action
            )
        )
    )
    3: Expr_Assign(
        var: Expr_Variable(
            name: file
        )
        expr: Expr_ArrayDimFetch(
            var: Expr_ArrayDimFetch(
                var: Expr_Variable(
                    name: _FILES
                )
                dim: Scalar_String(
                    value: file
                )
            )
            dim: Scalar_String(
                value: name
            )
        )
    )
)
```

The implementation for supporting this operation is shown in the Listing 3.31. As an array fetch expression in AST, its node type is “ArrayDimFetch”. It is constructed with

one “var” sub-node, which referenced the array itself, and one “dim” sub-node, which referenced which index of element will be fetched from this array.

First, we evaluate the “var” sub-node. If it is a variable node type that means the array is a one-dimensional array. Like the 2^{nd} and 3^{rd} statement in the Listing 3.29. And then, we further check whether it is a superglobal variable. If so, we get the content and treat it as a symbol value with the type “Symbol_SuperGlobal”. If not, we try to evaluate this array from the environment by its name. If this array was initialed in the environment, we try to fetch the element by the its index. The index is from the evaluation of sub-node “dim”. If the array is not initialed before it used, we assume a symbol value to represent it.

Listing 3.31: One Dimensional Array Fetch Evaluation

```
1  //deal with the 1D arrayDimFetch
2  if ( $t == "PhpParser\Node\Expr\Variable" ) {
3
4      $array_name = $var->name;
5
6      $array_index = $dim->value;
7
8      //If the array that fetch by the source code is not a ...
           superglobal array
9      if ( ! in_array( $array_name, $SG_Flag ) ) {
10         $result = array();
11
12         //Type of $node->var is "PhpParser\Node\Expr\Variable", and
13         //create ASNode type index to get the element from the ...
               ASNode array
14         if ( is_string( $array_index ) ) {
15             $indexASNode = new NodeString( $array_index );
16         } elseif ( is_int( $array_index ) ) {
17             $indexASNode = new NodeInteger( $array_index );
18         } else {
19             die( "We only concern the integer and string type of ...
                   index!" );
20         }
21
22         //Parsing the "$array->var" and getting sub-types of ASNode ...
               array
23         $dict_Array = eval_node( $var, $env, $layer + 1 );
24
25         foreach ( $dict_Array as $k => $v ) {
26             if ( $v instanceof NodeArray ) {
27                 $result[ $k ] = $v->getValueByKey( $indexASNode );
28
```

```
29            }  else {
30               $temp = new NodeSymbol_array( "Array_NewCase:" . ...
                     get_class( $v ) . "[$array_index]" );
31               $result[ $k ] = $temp;
32            }
33         }
34
35         return $result;
36      }
37   }
```

For the two-dimensional array fetch, the process is similar to the one-dimensional fetch but it gets the first dimensional element by a recursively call and fetch the second dimensional element by the index. The implementation for supporting the 2D array evaluation was showed in Listing 3.32. And the Listing 3.33 show symbolic execution result that is generated from the code in the Listing 3.29.

Listing 3.32: Two Dimensional Array Fetch Evaluation

```
1  if ( $t == "PhpParser\Node\Expr\ArrayDimFetch" ) {
2
3     if ( get_class( $dim ) == "PhpParser\Node\Scalar\String_" ) {
4        $index = $dim->value;
5
6     } elseif ( get_class( $dim ) == "PhpParser\Node\Expr\Variable" ...
         ) {
7        $index = $dim->name;
8
9     } else {
10       die( "New index in 2D arrayDimFetch" );
11    }
12
13    assert( is_string( $index ) );
14    $resultDict = eval_node( $var, $env, $layer + 1 );
15
16    $result = array();
17
18    foreach ( $resultDict as $k => $v ) {
19       if ( $v instanceof leafNodeArray ) {
20          $result[ $k ] = $v->getValueByKey( new leafNodeString( ...
               $index ) );
21
22       } else {
23          $temp_Index = new LeafNodeString( $index );
24          $temp = new opNodeArrayDimFetch( $v, $temp_Index );
25          $result[ $k ] = $temp;
26       }
```

```
27      }
28
29      return $result;
30  }
```

Listing 3.33: Global Environment

```
1  Path index: 1
2  Condition: None.
3
4  Environment:
5  arr => {(0=>abc:string) (1=>xyx:string) (2=>U.S.A:string) } : array
6  str => abc:string
7  var => (_REQUEST_action_symbol:symbol_SuperGlobal)
8  file => (_FILES_file_name_symbol:symbol_FILES)
```

3.3 Generating Z3 Satisfiability Constraints

For each path constraint, we need to automatically verify whether it is feasible. Meanwhile, our system will generate concrete inputs that will lead to the execution of a path. In order to accomplish this objective, our system translates a path constraint into a set of Z3 constraints. The Z3 syntax rules was shown in the table 3.1.

Table 3.1: Z3 Syntax Rules

Variable ::=	Z3 Statement
$\mid string$	"string"
$\mid Integer$	Integer
$\mid Boolean$	Boolean
$\mid Symbol$	(declare-const Symbol String)
Operation ::=	
$\mid Concat\ \ expr_1\ \ expr_2$	(str.++ $expr_1\ expr_2$)
$\mid Negate\ \ expr$	(not $expr$)
$\mid Empty\ \ expr$	(= (str.len $expr$) 0)
$\mid AND\ \ expr_1\ \ expr_2$	(and $expr_1\ expr_2$)
$\mid OR\ \ expr_1\ \ expr_2$	(or $expr_1\ expr_2$)
$\mid Equal\ \ expr_1\ \ expr_2$	(= $expr_1\ expr_2$)
$\mid NotEqual\ \ expr_1\ \ expr_2$	(not (= $expr_1\ expr_2$))
$\mid Greater\ \ expr_1\ \ expr_2$	($>$ $expr_1\ expr_2$)
$\mid Smaller\ \ expr_1\ \ expr_2$	($<$ $expr_1\ expr_2$)
Commands::=	
$\mid (declare - const\ \ a\ \ t)$	Declares a constant "a" of give type
$\mid (assert\ \ f)$	Adds a formula into Z3 internal stack
$\mid (check - sat)$	Check the formula on the Z3 stack are satisfiable or not
$\mid (get - model)$	Retrieve an interpretation that makes all formulas true.

Our Z3 interpreter will recursively interpret the tree structural path constraints to generated the Z3 assertion for each path. The Z3 interpreter starts to interpret the root of the tree, and generates the the Z3 assertion depended on the operation of the root. After that, the Z3 interpreter will recursively interpret the root's children nodes until the leaf nodes.

As an example in Figure 3.3, its root operation is a binary operation "AND". It has two

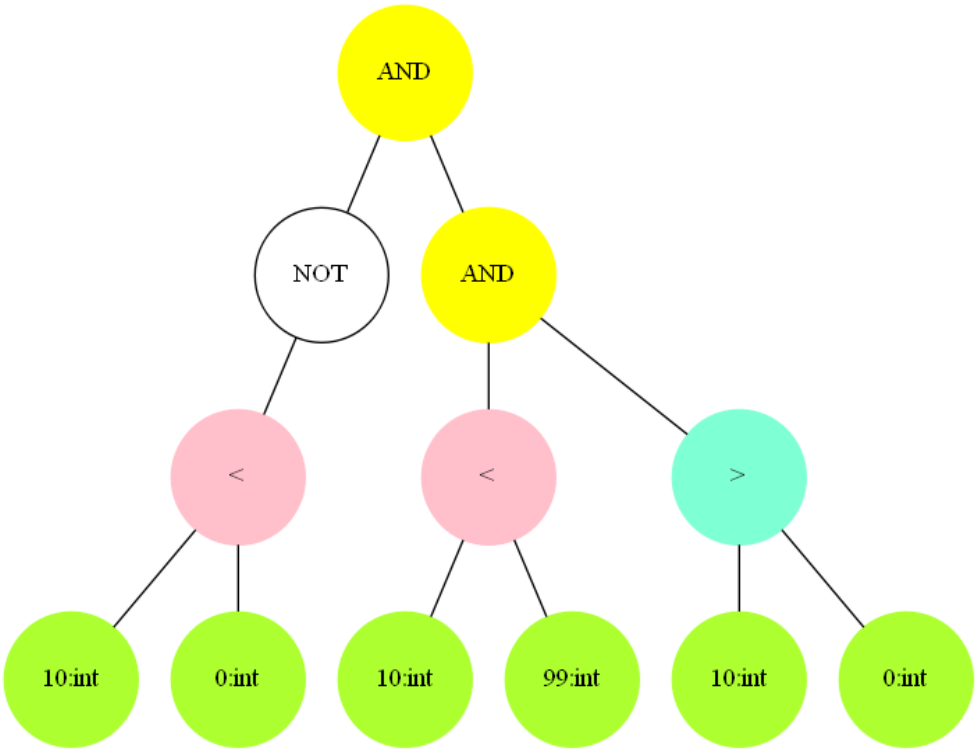

Figure 3.3: Condition Constraint

branches in its left and right. Firstly, the Z3 interpreter will generate the "AND" operation in Z3, $(and \;\; expr_1 \;\; expr_2)$, and recursively interpret the left branch and next the right branch. For the left branch interpretation, it is an unary operation "Negate" , which just has one sub-node. The Z3 interpreter will generate $(not \;\; expr)$ and continue interpret the "Smaller" operation in its sub-node. After the two leaf "Integer" nodes were interpreted, the recursive process in left branch will start to return to the root, and the current Z3 assertion for the example is $(and \;\; (not \;\; (< \;\; 10 \;\; 0)) \;\; expr_2)$. After the right branch was interpreted, the $expr_2$ will be replaced, and the final Z3 result for Figure 3.3 is

$$(and \;\; (not \;\; (< \;\; 10 \;\; 0)) \;\; (not \;\; (and \;\; (<= \;\; 10 \;\; 99) \;\; (> \;\; 10 \;\; 0)))).$$

In the Z3 internal, there is a internal stack to be used to check the "satisfiable" for all the asserted formulas by the user. If all the formulas are true. The Z3 will return "satisfiable" for inserted formulas. For the Z3 result about Figure 3.3, the Z3 return "satisfiable". So this path is feasible path in this PHP example.

Evaluation

In order to evaluate the effectiveness of our system, we have performed evaluation using both self-designed examples and PHP programs collected from production platforms.

4.1 A Running Example

As an example, the Listing 4.1 presents a PHP program that generates 6 paths in the global environment. And only one path is a "feasible" path. After this section, I will show how the Z3 solver help us to verify the "feasible" path.

Listing 4.1: PHP Codes Example

```
1  $a = 10;
2  $b = 0;
3  $c = '';
4
5  if ( $a < 0 ) {
6      $c = 'Failure';
7  }
8
9  if ( $a ≥ and $a > 0 ) {
10     $a = 100;
11     $b = 200;
12     $c = 'Failure';
13
14 } elseif ( !is_null($b) && $a > 10 ) {
15     $a = 666;
16     $c = 'Success!;
17
18 } else {
19     $c = 'Failure';
20 }
```

The conditional constraint for each path about the PHP code above is shown in the Listing 4.2. And the Z3 assertion for each path was shown in the Listing 4.3.

Listing 4.2: Conditional Constraint

```
1
2  Path index: 1
3  Condition: (operator: AND true:bool,(operator: AND (operator: NOT ...
       true:bool,),(operator: > ...
       (_Unknow_Argument_:symbol_String),0:int,),),)
4
5
6  Path index: 2
7  Condition: (operator: AND (operator: NOT true:bool,),(operator: ...
       AND (operator: NOT true:bool,),(operator: > ...
       (_Unknow_Argument_:symbol_String),0:int,),),)
8
9
10 Path index: 3
11 Condition: (operator: AND (operator: AND true:bool,(operator: NOT ...
       (operator: AND (operator: NOT true:bool,),(operator: > ...
       (_Unknow_Argument_:symbol_String),0:int,),),),),(operator: AND ...
       (operator: ≥ ...
       (_Unknow_Argument_:symbol_String),10:int,),(operator: > ...
       (_Unknow_Argument_:symbol_String),10:int,),),)
12
13
14 Path index: 4
15 Condition: (operator: AND (operator: AND (operator: NOT ...
       true:bool,),(operator: NOT (operator: AND (operator: NOT ...
       true:bool,),(operator: > ...
       (_Unknow_Argument_:symbol_String),0:int,),),),),(operator: AND ...
       (operator: ≥ ...
       (_Unknow_Argument_:symbol_String),10:int,),(operator: > ...
       (_Unknow_Argument_:symbol_String),10:int,),),)
16
17
18 Path index: 5
19 Condition: (operator: AND (operator: AND true:bool,(operator: NOT ...
       (operator: AND (operator: NOT true:bool,),(operator: > ...
       (_Unknow_Argument_:symbol_String),0:int,),),),),(operator: NOT ...
       (operator: AND (operator: ≥ ...
       (_Unknow_Argument_:symbol_String),10:int,),(operator: > ...
       (_Unknow_Argument_:symbol_String),10:int,),),),)
20
21
22 Path index: 6
23 Condition: (operator: AND (operator: AND (operator: NOT ...
       true:bool,),(operator: NOT (operator: AND (operator: NOT ...
       true:bool,),(operator: > ...
       (_Unknow_Argument_:symbol_String),0:int,),),),),(operator: NOT ...
```

```
    (operator: AND (operator: ≥ ...
    (_Unknow_Argument_:symbol_String),10:int,),(operator: > ...
    (_Unknow_Argument_:symbol_String),10:int,),),),)
```

Listing 4.3: Conditional Constraint

```
1  Path No.1
2  (assert (= (and (≤ 10 0) (and (≥ 10 0) (≤ 10 10))) true))
3  (check-sat)
4  (get-model)
5
6  Path No.2
7  (assert (= (and (not (≤ 10 0)) (and (≥ 10 0) (≤ 10 10))) true))
8  (check-sat)
9  (get-model)
10
11 Path No.3
12 (assert (= (and (and (≤ 10 0) (not (and (≥ 10 0) (≤ 10 10)))) ...
       (and (≥ 10 10) (≤ 10 100))) true))
13 (check-sat)
14 (get-model)
15
16 Path No.4
17 (assert (= (and (and (not (≤ 10 0)) (not (and (≥ 10 0) (≤ 10 ...
       10)))) (and (≥ 10 10) (≤ 10 100))) true))
18 (check-sat)
19 (get-model)
20
21 Path No.5
22 (assert (= (and (and (≤ 10 0) (not (and (≥ 10 0) (≤ 10 10)))) ...
       (not (and (≥ 10 10) (≤ 10 100)))) true))
23 (check-sat)
24 (get-model)
25
26 Path No.6
27 (assert (= (and (and (not (≤ 10 0)) (not (and (≥ 10 0) (≤ 10 ...
       10)))) (not (and (≥ 10 10) (≤ 10 100)))) true))
28 (check-sat)
29 (get-model)
```

We select the Path No.1 and Path No.4 as two instance to show how the condition constraint was converted to Z3 assertion, and how Z3 help us to verify the path. The constraint for Path No.1 was shown in the Figure 4.1.

From the Figure 4.1, we know that the Path No.1 is a "infeasible" path, because the left branch of the root is false, "$10 < 0$" is false, and the root is a "AND" operation. Therefore,

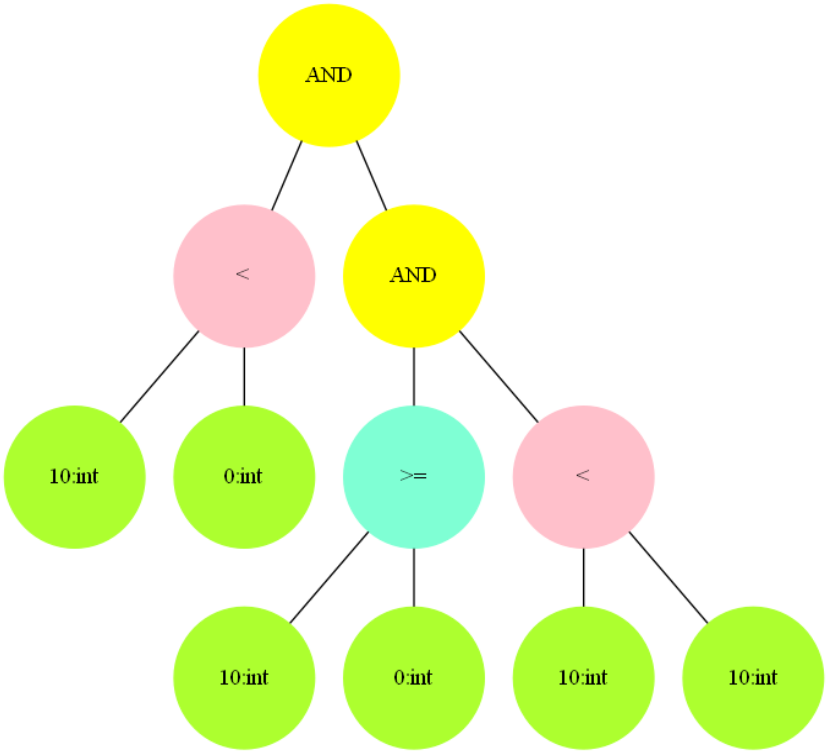

Figure 4.1: Conditional Constraint for Path index: 1

the result from the Z3 solver is “unsat” (unsatisfied) that means the Path No.1 is a infeasible path for the example in the Listing 4.1.

After go thought all Z3 assertions in the Listing 4.3, we know the assertion of Path No.4 is the only one satisfied path in the Listing 4.1 example. The Figure 4.2 shows the tree structure of the conditional constraint in Path No.4.

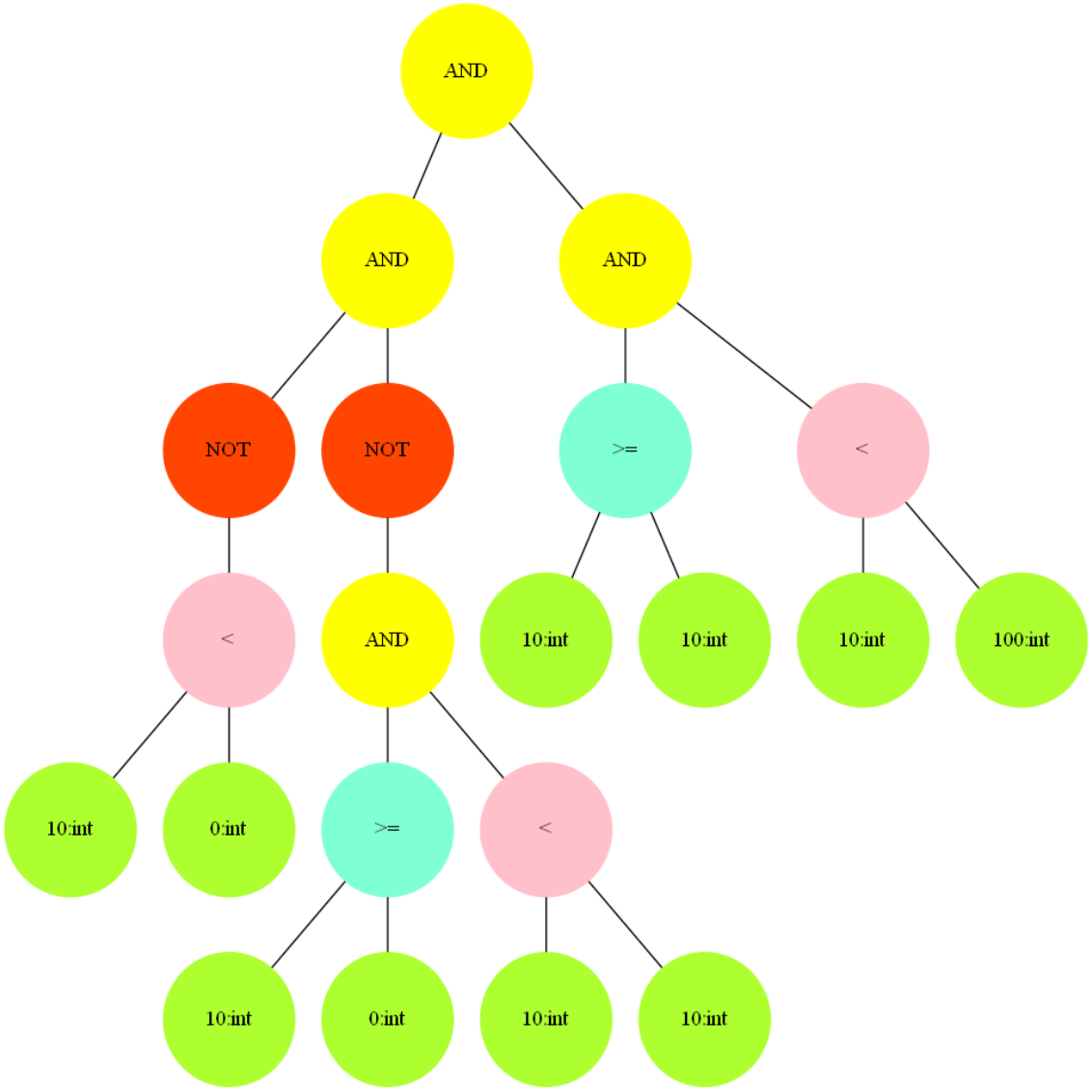

Figure 4.2: Conditional Constraint for Path index: 4

4.2 Experiments Using Real Examples

We have leveraged real-world samples collected from PHP code repositories including Wordpress plugins and Github. Our system can effectively perform symbolic execution for all the tested samples. Table 4.1 presents the name of WordPress Plugin, the version, the lines of code, and the time that is required to perform symbolic execution.

Table 4.1: Execution Result

Plugin Name	Version	Number of Path	Time(seconds)
Estatik	2.2.5	12	0.86
FoxyPress	0.4.1.1	65	0.86
Adblock Blocker	0.0.1	7	0.42
N-Media . . . Uploader	1.3.4	126	0.51
Advanced Ads.	1.8.4	256	0.49
Power Play	3.3	2448	0.71
Enbale_Media_Replace	3.0.6	374	0.58
WooCommerce-Catalog-Enquiry	3.0.0	1728	0.71

Conclusion

This thesis presents the design, implementation, and evaluation of an abstract-syntax-tree-oriented symbolic execution engine for the PHP programming language. As a symbolic execution engine, our system emulate the execution of a PHP program by assuming that all inputs are with symbolic rather than concrete values. While our system inherits the basic definition of symbolic execution, it fundamentally differs from existing symbolic execution implementations that mainly leverage intermediate representation (IRs) to operate. Specifically, our system directly takes the abstract syntax tree (AST) of a program as input and subsequently interprets this AST. Performing symbolic execution using AST offers unique advantages. First, it enables one-to-one mapping between the source code and the analysis results such as control flows and data flows. Second, it makes possible the direct instrumentation on source code to enable developer-aware changes. Third, it has higher applicability since IR is not always available. The design and implementation of our symbolic execution engine essentially feature an interpreter that interprets the AST based on symbolic values. Different from an interpreter that deterministically follows a single execution path by operating on concrete input values, the interpreter we have built needs to generate all paths, where each path has a constraint and its own environment. Constraints and environments of paths need to be dynamically created and maintained while the AST is evaluated. Our interpreter is context-dependent, where all user-defined functions are faithfully when they are called. Once all paths for a program is generated, we will automatically translate the constraint of each path into assertions that can be verified by

satisfiability modulo theories (SMT) solver (e.g., Z3). The SMT solver can further verifies assertions for each path and report i) concrete input values that enable this path or ii) the infeasibility of this path. We have tested our system using both prototype PHP programs and real PHP programs collected from WordPress plugins. The experimental results have demonstrated our system is highly effective in performing symbolic execution.

Bibliography

[1] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[2] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[3] Elliot J. Chikofsky and James H Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990.

[4] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[5] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering*, (3):215–222, 1976.

[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[7] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[8] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices*, 46(3):265–278, 2011.

[9] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[10] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*. Citeseer, 2009.

[11] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Conference on Computer Aided Verification*, pages 365–373. Springer, 1989.

[12] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

[13] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and privacy (SP), 2010 IEEE symposium on*, pages 317–331. IEEE, 2010.

[14] Robert S Boyer, Bernard Elspas, and Karl N Levitt. Selecta formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6):234–245, 1975.

[15] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[16] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[17] A PHP parser written in php. https://github.com/nikic/PHP-Parser. Accessed: 2017-05-1.

[18] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.

[19] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[20] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[21] WordPress Plugins – plugins extend and expand the functionality of your website. https://wordpress.org/plugins/. Accessed: 2017-05-1.