

MASTER'S THESIS

Determining paths to injection vulnerabilities in PHP-code Using Symbolic Execution

Dohmen, G.

Award date:
2022

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain.
- You may freely distribute the URL identifying the publication in the public portal.

Take down policy

If you believe that this document breaches copyright please contact us at:

pure-support@ou.nl

providing details and we will investigate your claim.

Downloaded from <https://research.ou.nl/> on date: 23. Jan. 2023

Open Universiteit
www.ou.nl



Determining paths to injection vulnerabilities in PHP-code

Using Symbolic Execution

G. M. L. Dohmen

Student:
Date: June 26, 2022

DETERMINING PATHS TO INJECTION VULNERABILITIES IN PHP-CODE

USING SYMBOLIC EXECUTION

by

G. M. L. Dohmen

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Science
Master Software Engineering
to be defended publicly on July 1, 2022 at 09:00 AM.

Student number:

Course code: IM9006

Thesis committee: Dr. Ir. H.P.E. Vranken (chairman), Open University
Prof. Dr. E.J. Vos (supervisor), Open University





For Monique, Emma & Marijn

ACKNOWLEDGEMENT

Working on a master's thesis is quite a job, and much more work when the ideas and possibilities seem to expand endlessly. This thesis presents the result of a limited, but extensive search.

My thanks go in the first place to my family: Monique, Emma and Marijn. In the beginning, they were not aware of the consequences of studying alongside a job, but fortunately, despite the strain, they were in many ways an incentive to keep on going with this study. Getting this opportunity, alongside family life and work, is not something that everyone can do or take for granted.

I would also like to thank my supervisor, dr. Ir. Harald Vranken, for his patience and valuable feedback. At the crucial moment, he gave me the push (and the opportunity) I needed to reach the finish line.

Finally, I would like to thank François Molin. During our studies, we were on the same wavelength, which led to many inspiring conversations.

SUMMARY

Vulnerabilities in software are common and increasingly maliciously exploited by hackers. In addition to dynamic testing of (compiled) code, static analysis of source code is an important aspect of improving software quality. Without running the program, we want to be able to measure the quality of the code. One of the quality factors of computer software is how the program will behave on a given input.

In this research, we sought a method to efficiently find a path that leads to a vulnerability. To do this we first need to find a potential path; a sequence of statements that goes from the entry point where the data is entered, to the exit point where a vulnerability can be triggered. Next we must determine which input causes the path to be executed. To determine that input, we use symbolic execution to find solutions.

When we execute a program symbolically, we do not use concrete input values, but leave them open. By expressing the program code as best we can in logical equations of these symbolic inputs, we can try to determine from the solutions of these logical equations which inputs will lead the hacker to vulnerabilities in the code.

In the study, we looked at a number of aspects in light of this context. How can we transform source code into a form in which we can find paths? How can we find these code paths in this search structure? How can we convert the code for these paths into a form that can be executed symbolically, and what limitations and solutions do we encounter when using this logical equation to determine the largest possible input that triggers these paths? Finally, we looked specifically at what language constructs a software tool must master in order to symbolically execute as many samples as possible from a commonly used test set.

For finding paths, we examined the extent to which PHP code is convertible to control flow graphs. For these CFGs, we designed an algorithm to search for paths between two nodes. We can control the search process because more precise targets can be specified.

The next aspect was the question of how to translate the PHP code of a code path into logical equations that allow it to be executed symbolically. A first problem here was that the dynamically typed language PHP, must be translated into the statically typed language SMTLIB. To this end, we proposed a type inference algorithm.

Of the huge number of functions that exist in PHP, only a very small number can be translated one-to-one into SMTLIB. This means that in many cases it is not possible to use symbolic execution to compute appropriate input values for a found. However, for functions that work with arrays, creative solutions appear to be possible.

If we apply the proposed methodology to PHP code, it appears that Z3 sometimes proposes a model as a solution that has little value. This happens, for example, when working with symbolic strings. Although Z3 can work with regular expressions, reasoning with symbolic regular expressions in Z3 does not succeed at all.

Using the Samate test set, we determined which language construct we must be able to translate to SMTLIB in order to fully execute a sample from the test set symbolically. This could point a direction in which further research could be conducted.

CONTENTS

List of Figures	vii
List of Tables	ix
Acronyms	x
1 Introduction	1
1.1 Thesis Outline	4
2 Preliminaries	5
2.1 Determining input values that trigger paths to injection vulnerabilities in PHP-code using Symbolic Execution.	5
2.2 The PHP programming language	5
2.3 Vulnerabilities	10
2.4 Static Software analysis.	12
2.5 Paths in software.	15
2.6 Triggering paths	18
2.7 Symbolic Execution	20
2.8 Static Single Assignment (SSA)	24
2.8.1 branches	24
2.8.2 loops	26
2.9 SMT solvers.	27
3 Research Design	29
3.1 Research questions	29
3.2 Research method	30
3.3 selection of frameworks and software tools.	31
3.3.1 Analysis of existing tools for creating AST's.	32
3.3.2 Choosing appropriate programming languages.	33
3.3.3 Choosing an appropriate SMT solver	34
3.4 Testing.	37
3.5 Data description.	37
3.5.1 Test files with isolated PHP code	37
3.5.2 PHP Vulnerability test suite	37
4 Finding Paths	39
4.1 RQ1: How can we find paths in PHP-code efficiently?	39
4.2 Create AST's from PHP-files	39
4.3 More about ASTs.	43
4.4 Create a searchable data structure from an ast-file	45
4.4.1 Algorithm B	46
4.4.2 Statements	47
4.4.3 Assignments	47

4.4.4	The WHILE statement	48
4.4.5	The IF statement	50
4.4.6	Switch	53
4.4.7	Functions	54
4.4.8	Functions calling functions	57
4.4.9	Functions Variables	58
4.4.10	Array's	59
4.4.11	Graphs resulting from algorithm B	62
4.5	Construction of paths	63
5	Evaluating PHP code by Symbolic Execution	68
5.1	RQ2: Up to what extent can PHP-code be evaluated using Symbolic Execution?	68
5.2	Variables in SMTLIB	68
5.3	Conditions and assignments in SMTLIB	72
5.4	Array's	73
5.4.1	Single variables	73
5.4.2	Using SMTLIB's array's	75
5.5	Functions	78
5.5.1	User defined functions functions	78
5.5.2	Function return values	79
5.5.3	String functions	79
5.5.4	array functions	81
5.6	Type inference	84
6	Solving Path Conditions	89
6.1	Draw up path conditions	89
6.2	Solving constraints with Z3	90
6.3	SAMATE	91
6.3.1	The size and complexity of the dataset	91
6.3.2	Usability	92
6.3.3	Code coverage	94
6.4	Specific code fragments	98
6.4.1	Regular expressions	99
6.4.2	String functions	100
6.4.3	Numeric strings	100
7	Conclusion	102
7.1	Discussion	102
7.1.1	RQ1: How can we find paths in PHP-code efficiently?	102
7.1.2	RQ2: Up to what extent can PHP-code be evaluated using Symbolic Execution?	103
7.1.3	RQ3: Up to what extent can path conditions be solved?	103
7.2	Research contribution	103
7.3	Limitations and future work	104
7.4	Related work	105
7.4.1	Finding paths in a graph	105
7.4.2	Symbolic Execution	105

7.4.3	Tools resulting from previous research	106
7.4.4	Research related to SMT solvers	107
A	Example of PHP code vulnerable to injection	108
B	Example of how a SAMATE PHP file translates into SMTLIB	109
C	Comparison of PHPAST and PHPLY	110
D	Operators for types in SMTLIB	113
E	Utilities	116
E.1	SAMATE line number inserter.	116
E.2	SAMATE CWE selector	116
E.3	Remove special characters from file name	117
E.4	Batch file creator.	118
E.5	Testing tool for graph unit	118
E.6	Tool for extracting AST functions	119
E.7	Testing tool for type inference unit.	119
E.8	Testing tool for testing AST functions	120

LIST OF FIGURES

2.1	Traditional use of PHP on a web server	6
2.2	Tiobe index, April 2022	6
2.3	Overview of a Stored XSS injection attack [54]	11
2.4	Overview of an SQL injection attack [47]	12
2.5	AST of the FOR-statement for($i=0$; $i<4$; $i++$)	13
2.6	Example program with control flow graph	14
2.7	Structure of a PHP program	17
2.8	Structure of a PHP program with classes	17
2.9	Example of a PHP program and the accompanying CFG [35]	18
2.10	Overview of variable collections used in path code. The intersection of the sets R and V can be empty.	19
2.11	Symbolic execution tree	22
2.12	Example of the use of the ϕ function in a basic block after a branch.	25
2.13	Example of the use of the ϕ function in while loop, which contains a branch	27
3.1	Global overview of the stages of our research	31
3.2	Design matrix for the tool	32
3.3	Initial design of the tool (PC: Path Conditions, SC: Security Conditions)	35
3.4	Final (adapted) design of the tool	36
4.1	Excerpt from an AST file showing function declarations and function calls	43
4.2	Example of function declarations and function calls in an AST	44
4.3	The red nodes form a subtree	44
4.4	ASTs with multiple function calls	44
4.5	The yellow node is the lowest common ancestor (LCA) of the red and blue node.	46
4.6	In an AST the LCA is a AST_STMTLIST node	46
4.7	A single assignment in a CFG for a single expression <i>simple block</i>	47
4.8	In our algorithm assignments in a CFG are a series of nodes for partial calculations	48
4.9	The arrows after a WHILE statement are labelled with the conditions.	48
4.10	The conditions of a WHILE statement are represented by extra nodes before the statement in the loop and after the WHILE	49
4.11	Nodes in a CFG resulting from an IF statement	50
4.12	Conditions for branches of a conditional statement in a CFG are inserted as extra nodes before the statements.	52
4.13	CFG of a Switch statement with all branches closed with a BREAK	53
4.14	CFG of Switch statements with 1 or 2 omitted BREAK statements.	53
4.15	A node with a function call, calling the function $f(a,b)$ with the arguments x and y	56

4.16 Example of a CFG where the function call is replaced by a parameter substitution and the CFG of the function.	57
4.17 Example of a CFG constructed with algorithm B	62
4.18 On the left: a small CFG to illustrate the search algorithm. On the right: the resulting search tree.	65
5.1 Symbolic execution tree with SSA	71
5.2 adapted PHP type tree	84
E.1 Utility for preparing a subset of the SAMATE dataset	117
E.2 Utility for removing special characters from filenames	117
E.3 Utility for creating a batch file that produces all AST files	118
E.4 Unit test for the Graph unit	118
E.5 Tool for extracting AST nodes from AST files	119
E.6 Unit test for the type inference unit	119
E.7 Tool for testing various AST functions	120

LIST OF TABLES

2.1	Type juggling of strings and booleans to an integer or float	8
2.2	Type juggling of integers, floats and booleans to strings	8
2.3	Values that Type juggling converts to the boolean value false	8
2.4	Definitions of important points in the path of an injection vulnerability	15
2.5	fuzzing versus symbolic execution	23
2.6	standard code of a basic block converted to SSA form	24
2.7	standard code of a branch converted to SSA form	25
2.8	standard code of a while loop converted to SSA form	26
3.1	Characteristics of modern SMT solvers	35
3.2	Number of safe and unsafe samples for all CWE categories present in the SA-MATE testset.	38
5.1	Comparison of PHP and SMTLIB Types	69
5.2	Map PHP types to SMTLIB types	69
5.3	Example of a translation of PHP code including a function to SMTLIB	88
6.1	Examples of how line 6 sanitizes the input	92
B.1	Example of a translation of PHP to SMTLIB	109
C.1	List of AST nodes for PHPAST and PHPLY	111
C.2	List of AST nodes for PHPAST and PHPLY	112

ACRONYMS

AST	Abstract Syntax Tree
BFS	Breadth First Search
CFG	Context Free Grammar
CFG	Control Flow Graph
CG	Call Graph
DAG	Directed Acyclic Graph
DFS	Depth First Search
ICFG	Interprocedural Control Flow Graph
PDG	Program Dependency Graph
SA	Symbolic Algebra
SAT	Boolean Satisfiability Problem
SSA	Simple Static Assignment
SCC	Strongly Connected Component
SE	Symbolic Execution
SMT	Satisfiability Modulo Theory

1

INTRODUCTION

We expect that software works exactly as advertised. Especially now that software has gained a crucial, omnipresent and often invisible role in society, expectations increase that software performs reliably. A crashing program cannot only cause economical damage. People can be endangered - for example, when a storm surge barrier happens to be unreliable [34]. Researchers, but also the man on the street and politicians, have become aware that it is questionable whether all software can meet these high expectations.

Software engineers aim to deliver software with high quality. To achieve this quality, metrics are used to measure to what extent the software meets the quality requirements. Although there is no consensus on what an exact definition of software testing is, it is a point of view to see testing as the art of measuring the quality of software, according to these metrics. The outcome is not only a judgment according to some scale, but also a pointer to imperfections. Possibly with a remedy to improve the software [38].

There are many causes why software is not as perfect as we hoped [42]. Moreover, these imperfections are usually not noticeable, but surface under very specific conditions, which are difficult to find [39].

The eye of the master is still important in finding these errors. However, software engineers are increasingly supported by software tools. The human being can be relieved, because the search for errors can be automated. These tools are tireless and accurate. Moreover, they can not only search for known error sources, but also detect previously unknown error sources, for example by using artificial intelligence [19]. An increasingly important source of errors has to do with the security of the software. Secure software means that confidentiality, integrity and availability are guaranteed. To do this automatically, it is important that these conditions are translated into verifiable requirements [2].

The security of software can be compromised by its use in ways that were not intended. An unknowing user may have more use or access rights, which can inadvertently cause this user to cause damage. Software may also be vulnerable to abuse by malicious users.

It is known which concrete vulnerabilities have been reported¹. These alerts help the developers of the relevant software package to keep their software as secure as possible.

¹<https://cve.mitre.org/cve/>

In addition, vulnerabilities have been found that have not yet been made public by the researchers. Malicious hackers keep this knowledge secret in order to exploit the vulnerabilities. Ethical hackers usually agree with the developers that they will not make the discovery public until the vulnerability has been addressed.

We also know which categories of vulnerabilities are frequently found². This list can make developers aware of the fact that vulnerabilities are a common problem. But it does not prevent known vulnerabilities ending up in the software-to-be. It certainly doesn't provide developers with tools to prevent security flaws in software.

At the top of the OWASP list are the injection vulnerabilities. SQL injection is an example of this category. Injection vulnerabilities have in common that data is entered into the application, after which the data runs through a path through the software to a point where the data triggers a malicious action. In the case of SQL injection, a malicious SQL query is constructed from the input. Cross Site Scripting (XSS) is also in the OWASP top 10 as a separate category. Because here data entered elsewhere in the code triggers a vulnerability, this is de facto an injection vulnerability as well. With XSS, the data ends up in the output for another client. In the case of a Stored XSS, the data is stored in a database. Whenever data from the database reaches a user, the user is vulnerable because code is executed in addition to the static data the user expects. This code could, for example, send data, retrieve new code and so on.

Not all input triggers a vulnerability. Three conditions must be met for a vulnerability to be exploited. Firstly, it must be possible to enter user data into the software. Next, the code must follow a code path from the point where the data enters the program flow to the point where the vulnerability can be exploited. This is only possible if the input ensures that the conditions of control statements are such that the path to the vulnerability is followed. Finally, the value of variables and parameters at the endpoint must be such that the vulnerability is actually activated. In principle, a software engineer could manually analyze the software to determine the conditions under which a certain path can be traversed. However, the number of paths can be very large and certain paths can only be traversed with very specific input. A tool could help software engineers to check under which conditions a path can be executed.

Our research focuses on software written in the programming language PHP, as this is a widely used³ language for developing server-side software. In addition, our research is in line with previous research by, among others, Beisicht [6], which also focused on PHP code.

In our research we want to find out which approach can be used to activate the code paths leading to a vulnerability. To do this, we first need to find these paths, but we also need to find out with which input the program actually executes such a path.

In previous research, different ways of finding the right input were considered. One way is fuzzing [22]. An automated tool chooses input, hoping that this input leads to a place where the input triggers a software error or a vulnerability. The input can be random, or deliberately invalid, malformed or otherwise unexpected. Our research focuses on Symbolic Execution. A method that has been under research for some time [29]. The hope is that the input that triggers the vulnerability can be calculated, rather than having to guess at it.

²<https://owasp.org/www-project-top-ten/>

³<https://www.tiobe.com/tiobe-index/>

Therefore, the research question of our research is as follows:

Research Question: *How can we generate input that triggers paths on which injected data can be propagated in PHP-code using symbolic execution?*

1.1. THESIS OUTLINE

In chapter 2 we look at the background of the research. The design of the research is explained in chapter 3. The outcomes of the research per research question are presented in chapters 4, 5 and 6. The results of the research per research question are discussed in chapter 7, next to a description of the research contribution, limitation of the research and recommendations for future work.

2

PRELIMINARIES

2.1. DETERMINING INPUT VALUES THAT TRIGGER PATHS TO INJECTION VULNERABILITIES IN PHP-CODE USING SYMBOLIC EXECUTION

In this chapter, we provide background information. In section 2.2, we introduce the PHP programming language. In section 2.3, we discuss software vulnerabilities in general. In particular, we look at Injection vulnerabilities, which are the reason for this research. In section 2.4, we explain what static software analysis entails. In Section 2.5 we introduce the principle of a path through software. In Section 2.6 we look at the conditions that determine whether a path can actually be followed. In Section 2.7 we look at what Symbolic Execution (SE) is and how we can use the mathematical conditions that SE entails to determine under what conditions a path is passable. We will use Single Static Assignment (SSA) later in our research. We explain what SSA is about in section 2.8. Finally, in Section 2.9, we touch upon how specific computer software, SMT solvers, can be used to solve the mathematical conditions that SE entails.

2.2. THE PHP PROGRAMMING LANGUAGE

PHP is a scripting language that was initially developed for web servers. The PHP code is called through web pages. The code is interpreted by the parser and can output (part of) an HTTP response in the form of a web page, or another file type, such as an image. PHP scripts can connect to databases to retrieve or store data from databases using SQL queries (See figure 2.1). An example of (unsafe) PHP code is shown in Listing 2.1.

```
1 $username = $_POST[ 'user_input' ];  
2 mysql_query( "INSERT INTO 'table' ( 'username' ) VALUES ( '$username' )" );
```

Listing 2.1: Example of PHP code that allows SQL injection, because the input has not been sanitised

The first version of PHP saw the light of day in 1994. It was not until twenty years later, in 2014, that a more formal specification for PHP was proposed. A recent specification of the PHP syntax in EBNF form can be found here:

https://github.com/php/php-src/blob/master/Zend/zend_language_parser.y

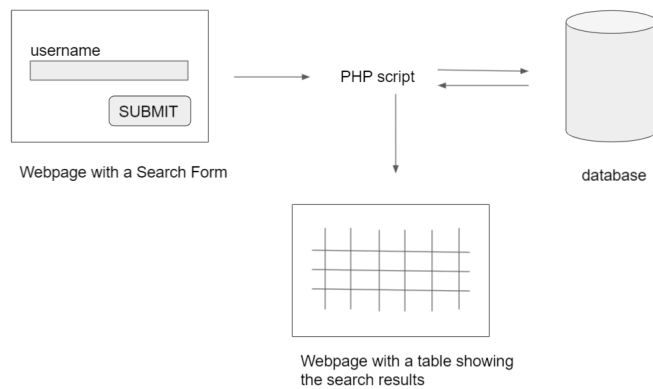


Figure 2.1: Traditional use of PHP on a web server

The history page of this web page shows the changes up to October 2000. Sometimes a commit shows which language construct was tinkered with, but usually there is no information about the PHP version in which the changes were introduced.

PHP can also be run from the command line to run scripts on a (local) computer. Furthermore, there are nowadays compilers that compile PHP code into executable code, e.g. PeachPie¹. PHP is being actively developed and currently (April 2022) ranks tenth on the Tiobe-Index of most-used programming languages (See figure 2.2).













Apr 2022	Apr 2021	Change	Programming Language	Ratings	Change
1	3	▲	 Python	13.92%	+2.88%
2	1	▼	 C	12.71%	-1.61%
3	2	▼	 Java	10.82%	-0.41%
4	4		 C++	8.28%	+1.14%
5	5		 C#	6.82%	+1.91%
6	6		 Visual Basic	5.40%	+0.85%
7	7		 JavaScript	2.41%	-0.03%
8	8		 Assembly language	2.35%	+0.03%
9	10	▲	 SQL	2.28%	+0.45%
10	9	▼	 PHP	1.64%	-0.19%
11	16	▲	 R	1.55%	+0.44%
12	12		 Delphi/Object Pascal	1.18%	-0.29%

Figure 2.2: Tiobe index, April 2022

PHP has a number of special language features. The online documentation explains these features, but often incompletely. Changes in the latest versions of PHP (approximately from version 7.0) are more clearly documented by explaining the changes from the previous version .

¹<https://www.peachpie.io/>

Dynamic Typing. In PHP, variables do not have a fixed type, as is the case in Java, for example. Successive assignments can change the value and thus the type of a variable. Listing 2.2 provides a simple example of this.

```
1 $x=1;
2 echo $x;           // integer type
3 $x="hello";
4 echo $x;           // string type
```

Listing 2.2: The type of the variable \$x changes from integer to string

Which type a variable has at a certain point (and moment) may depend on the input. To determine that type, a procedure called type inference is needed to determine the type of an expression. An algorithm for type inference was first developed for functional languages. Robinson gave an algorithm for unifying logical expressions [43], which was later further developed by Hindley [26] and Milner [37] so that it could be used to determine the type of expressions. The Hindley Milner algorithm sometimes does not give an exact type, but it always gives a superset of the type.

If we look at the function maximum in listing 2.3, there are several possibilities for the type of variables \$a and \$b. The inequality operator > and the assignment operator = are defined for various types. For example, the function can be used to determine the maximum of two integers, but also for two strings (lexicographic ordering).

```
1 function maximum($a,$b){
2     if ($a>$b) {
3         $max=$a;
4     } else {
5         $max=$b;
6     }
7     return $max;
8 }
9
10 echo(maximum("hello","world")."\n");
11 echo(maximum(3,7));
```

Listing 2.3: The function maximum will work for different types (numberType and StringType, in this case)

Type Juggling². If an operand is not of the correct type, PHP tries to cast the type automatically. When PHP expects two numeric operands a string or boolean will be converted to an integer or float.

expression	interpreted as	outcome
1+'2'	1+2	3
1+'2.3'	1+2.3	3.3
1+'2.3e4'	1+2300	2301
1+'2text'	1+2	3
1+'2.3text'	1+2.3	3.3
1+'text2'	1+0	1
1+true	1+1	2
1+false	1+0	1

Table 2.1: Type juggling of strings and booleans to an integer or float

When concatenating, PHP expects two string values. Therefore PHP will try to cast integers, floats and boolean values to strings.

expression	interpreted as	outcome
'text'.1	'text'.1'	'text1'
'text'.1.23	'text'.1.23'	'text1.23'
'text'.true	'text'.1'	'text1'
'text'.false	'text'.	'text'

Table 2.2: Type juggling of integers, floats and booleans to strings

When PHP expects a Boolean value (for instance in a Boolean expression of an IF statement), it will treat all values in table 2.3 as false. All other values are treated as true.

value	type	treated as
false	boolean	false
0	integer	false
0.0	float	false
'0'	string	false
"	string	false
array[]	string	false
null	string	false

Table 2.3: Values that Type juggling converts to the boolean value false

²<https://www.php.net/manual/en/language.types.type-juggling.php>

Arrays³. In PHP, arrays are an ordered map that associates a key of type integer or string with a value. If the key is not of type integer or string, type juggling is used to make an integer of the key [20].

```
1 $x = array("key1" => "a", 3.14 => "b");
```

Listing 2.4: type juggling in an array

A second special feature is that a value can be assigned to an array without using an index. The highest integer index+1 is then automatically used. In the example in listing 2.5, the array is filled in line 1. The highest integer index is 3, because the decimal value 3.14 is converted to the integer 3 via type juggling. The first 'free' integer index in line 2 is then index 4.

```
1 $x = array("key1" => "a", 3.14 => "b");
2 $x[] = "c";
3 echo $x[4]; // the character "c" is shown
```

Listing 2.5: When assigning a value to an array without using an index, the index is automatically chosen

Aliasing and references⁴. The same variable can be accessed by another variable via assignment by reference. Aliasing makes static software analysis hard, because the order in which assignments are made matters.

```
1 $x = 0;
2 $y = &$x; // $y is an alias for $x
3 $y = "hello"; // assigning a value to $y, changes the value of $x
4 echo $x; // the text "hello" is shown
```

Listing 2.6: creating an alias for a variable

Variables can be used to name a variable. In listing 2.7 the value of variable \$x is used in line 3 by putting an extra \$ in front of the variable \$x.

```
1 $x = "a";
2 $a = "hello";
3 echo $$x; // the value "hello" is shown
```

Listing 2.7: variable variables

³<https://www.php.net/manual/en/language.types.array.php>

⁴<https://www.php.net/manual/en/language.variables.variable.php>

2.3. VULNERABILITIES

Software can be vulnerable to malicious attacks, allowing the attacker to use the software in an unintended way. The consequence of the attack may be that the software becomes unavailable for use (Denial of Service). This may be caused by the software going into a perpetual loop, or by using so many resources (such as memory or external storage) that the software effectively stops working. This type of attack is used by script kiddies, but also for political or ideological reasons. Professional attackers are often after data (e.g. email addresses) or credentials, which can then be used in later attacks. It is then not in the attacker's interest that the system is brought down. On the contrary, it is important that the attackers can go about their business unnoticed. This type of attack is complex and sometimes only noticed after months or years⁵.

```
1 // if an attacker can set the variable $attack equal to 1
2 $attack = $_POST("attack");
3 $playlist=array();
4 $x=random_int(0,9);
5
6 // the attacker can prevent the termination of the WHILE loop
7 while ($x not in playlist) || ($attack==1)
8 {
9     array_push(playlist, $x );
10    $x=random_int(0,9);
11 }
12 // never reached when $attack==1
```

Listing 2.8: The variable \$Attack can prevent that the loop will terminate

If an attacker exploits software, he must ensure that the program is executed in an unintended way. It may be that the program performs (a series of) actions that should not be possible. For example, authentication may be bypassed. An important feature of these attacks is that the attacker's input causes the software to perform the malicious actions. There is a path through the code between the point where the input (data) enters the code and a point further down the code where the data triggers the vulnerability. In listing 2.8 we see a program whose WHILE loop will never end if the variable \$attack gets the value 1.

The types of vulnerability are divided into categories by Mitre^{6,7}, called Common Weakness Enumerations (CWEs). The impetus for our research lies in CWEs occurring in the SAMATE test suite, which are related to injection and cross-site scscription vulnerabilities:

- CWE 78 : OS Command Injection⁸
- CWE 79 : Cross-site Scripting⁹
- CWE 89 : SQL Injection¹⁰

⁵<https://tweakers.net/nieuws/162686/ransomware-infecteerde-ook-back-upserver-van-universiteit-maastricht.html>

⁶<https://www.mitre.org/>

⁷https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

⁸<https://cwe.mitre.org/data/definitions/78.html>

⁹<https://cwe.mitre.org/data/definitions/79.html>

¹⁰<https://cwe.mitre.org/data/definitions/89.html>

- CWE 90 : LDAP Injection¹¹
- CWE 91 : XML Injection¹²
- CWE 95 : File Injection¹³
- CWE 98 : PHP Remote File Inclusion¹⁴

In a Cross Site Scripting attack, the attacker manages to get code into the program's data stream. This could be done by entering JavaScript code in an input field instead of normal data. This input is then processed as data elsewhere in the code, but can perform an action at a place where the program expects to process passive data.

For example, the input is displayed on another web page. If the input was JavaScript code, that code is executed by the web browser. The injected XSS code can also be stored, for example in a database. The injected code can then be executed again each time the program expects to request data. This is shown schematically in figure 2.3.

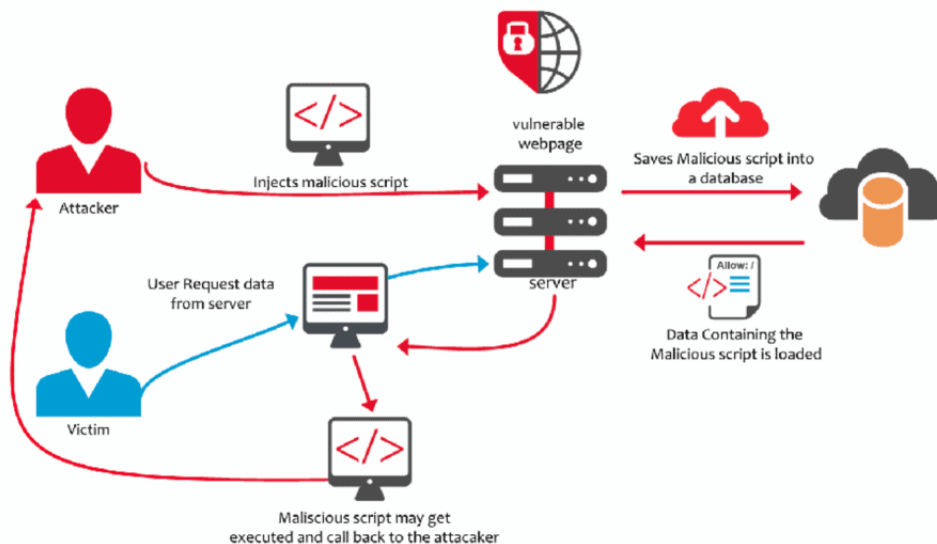


Figure 2.3: Overview of a Stored XSS injection attack [54]

With an SQL Injection vulnerability, the input is not or only partially sanitised, allowing an attacker to falsify SQL queries. For example, instead of just text (like a booktitle or a person's name), the attacker can add code to the input so that the query does not have the expected effect. For example, the query may retrieve not one but all lines from a database. This is illustrated in figure 2.4

¹¹<https://cwe.mitre.org/data/definitions/90.html>

¹²<https://cwe.mitre.org/data/definitions/91.html>

¹³<https://cwe.mitre.org/data/definitions/95.html>

¹⁴<https://cwe.mitre.org/data/definitions/98.html>

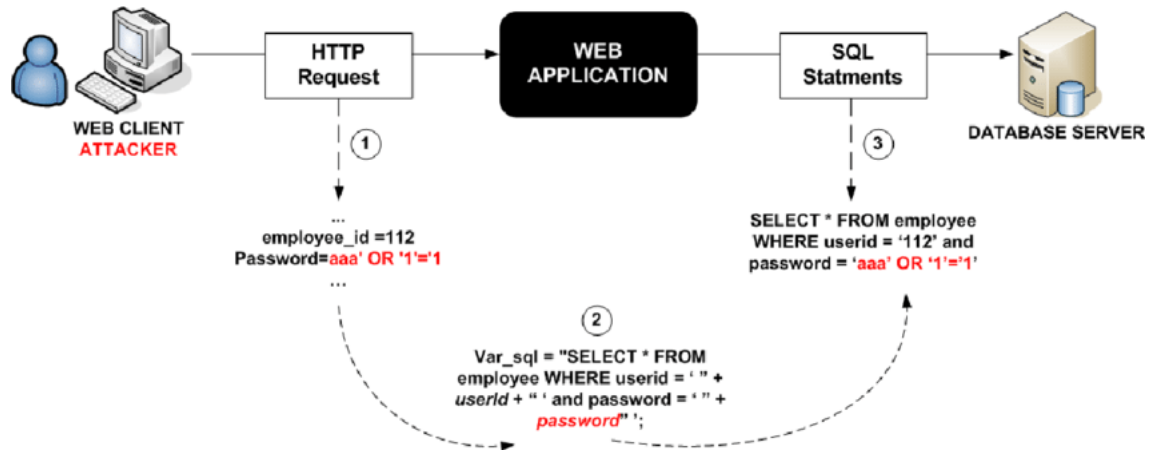


Figure 2.4: Overview of an SQL injection attack [47]

2.4. STATIC SOFTWARE ANALYSIS

If we want to find vulnerabilities in software, we can analyse the source code. We can manually read through the code and look for characteristic aspects of a piece of code. We can look at what conditions appear in the code, what function calls are present and how conditions affect the code flow. For large software projects, this method is not realistic because of the size and complexity. Software engineers therefore benefit from a software tool that performs this analysis. To do this, the tool must be able to read the code in a way that is usable by an automatic analyser. In the subsequent part of this section, we will look at ways in which the code can be presented in such a way that it is usable by automatic analysers.

ABSTRACT SYNTAX TREE

When an interpreter or compiler [1] reads source code, the lexer first determines the tokens. These are the smallest meaningful units within the Context Free Grammar (CFG) of that language. Examples of tokens in PHP are keywords (such as FOR, IF,...), identifiers of variables, literals, commas as separators and semicolons as terminals of a statement.

The parser creates an Abstract Syntax Tree (AST) from these tokens. In an AST, certain elements of the grammar can be omitted, because the structure of the syntax tree shows the relationship between the tokens. For example, an AST does not contain separators and terminal symbols such as commas and semicolons. As an example, all elements of the AST representing the statement `for(i=0; i<4; i++)` are shown in figure 2.5.

In an interpreter, the AST is used to execute the code. However, an AST can also be the starting point for forming a graph of the code.

CALL GRAPH

Informally, a Call Graph (CG) [23] represents which function is called by other functions in the code. Formally, a CG can be defined as follows (first we give the definitions for a Simple Graph and a Directed Graph).

Definition 2.4.1 (Simple Graph). A simple graph is an ordered pair $G = (V, E)$. With V being a set of vertices and E being a set of unordered pairs of distinct elements of V .

Definition 2.4.2 (Directed Graph). A directed graph $G(V, A)$ consists of the set V of the

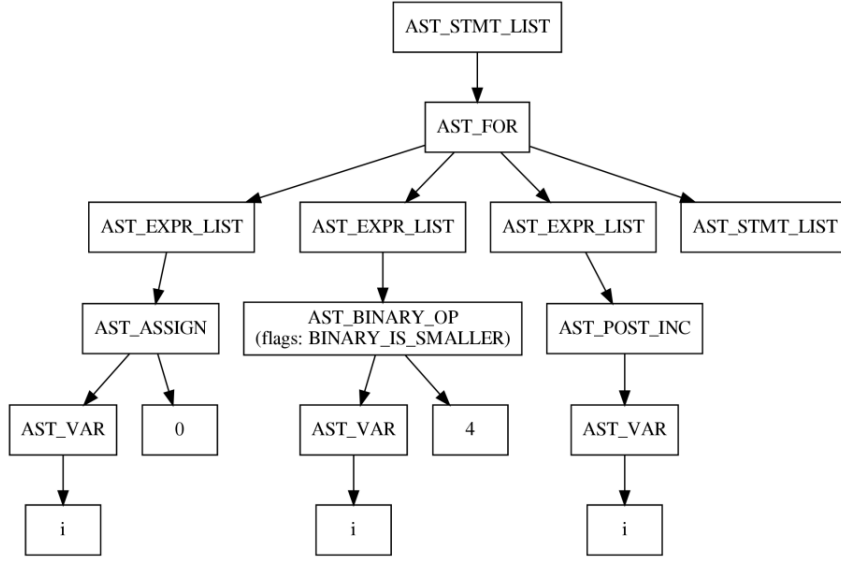


Figure 2.5: AST of the FOR-statement for(i=0; i<4; i++)

vertices v_i and the set A of the arrows (v_i, v_j) , where the ordered pair (v_i, v_j) is the arrow from vertex v_i to vertex v_j .

Definition 2.4.3 (Head, tail). Each arrow $a \in A$ in a directed graph has a head $h(a) \in V$ and a tail $t(a) \in V$. Thus each arrow leads from $h(a)$ to $t(a)$. The set of incoming arrows for a vertex $v \in V$ is defined as $in(v) = \{a \in A : t(a) = v\}$. Likewise the set of outgoing arrows $out(v) = \{a \in A : h(a) = v\}$.

Definition 2.4.4 (Neighbour). In a directed graph G a vertex v_j is a neighbour of vertex v_i if the arrow (v_i, v_j) is element of A .

Definition 2.4.5 (Path). A path in the directed graph $G(V, A)$ is an ordered n -tuple (v_0, v_1, \dots, v_n) where for each pair of consecutive vertices in the path the ordered pair (v_i, v_{i+1}) occurs in the set A (v_{i+1} is a neighbour of v_i). An empty path is the 0-tuple $()$. A path with exactly one vertex (v_i) has no arrow. A path can contain cycles, i.e. a vertex occurs more than once in such a path.

Definition 2.4.6 (Call Graph). A call graph is a static representation of the dynamic invocation relationships between procedures, function or methods (hereinafter referred to as function for short) in a program. A node in the call graph represents a function, and an edge $p \rightarrow q$ exists if function p can invoke function q . If a function p calls the function q more than once, we nevertheless represent it with a separate edge.

CONTROL FLOW GRAPH

A control flow graph (CFG [28]) is a directed graph in which all code of a procedure is included [35]. Burgstaller [9] list three forms to represent a CFG. For our research, we adapted the variant shown in figure 2.6:

- Nodes represent a *basic block*, a sequence of statements with one entry point (the first statement in the block) and one exit point (the last statement in the block).

- Basic blocks are connected by one or more arrows.
- Edges (arrows) are labeled with the condition that determines whether this transition may be taken.

Often, there are two special blocks: the entry block, through which control enters the flow graph, and the exit block, through which all control leaves [55]. Because nodes A of statements with $in(A) = 1 \wedge out(A) = 1$ are merged into a simple blocks, each arrow $A \rightarrow B$ has the property that $in(A) > 1 \vee in(B) > 1$ [50].

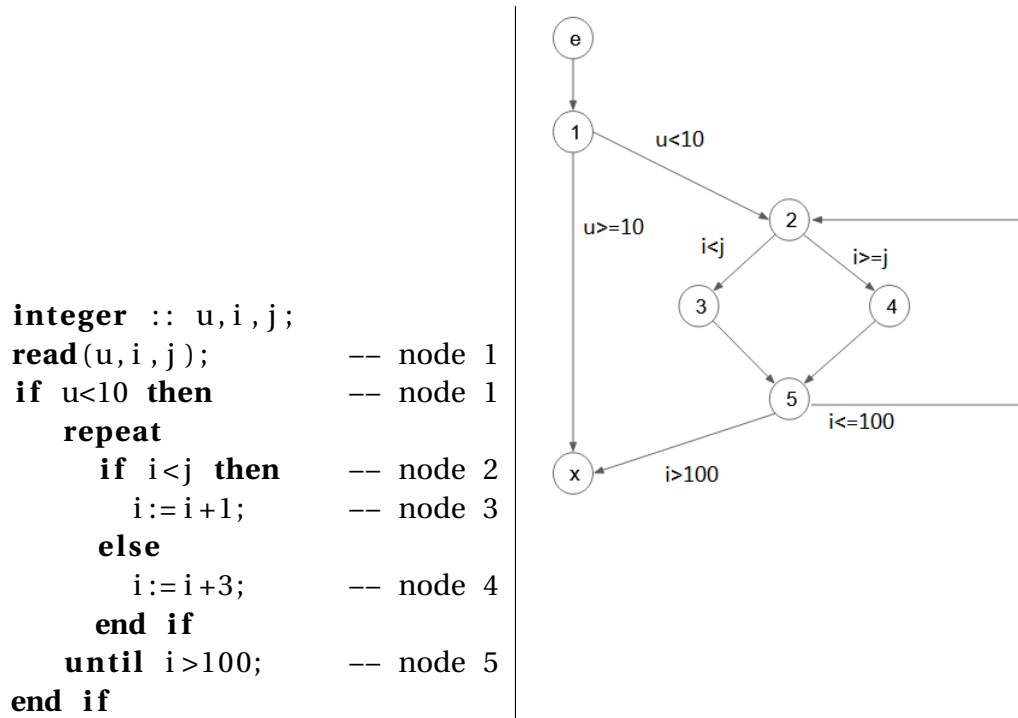


Figure 2.6: Example program with control flow graph

INTERPROCEDURAL CONTROL FLOW GRAPH

Control Flow Graphs only show the structure of a single function or procedure. To show the structure of a complete program, the subgraphs of the individual CFGs can be combined into an Interprocedural Control Flow Graph (ICFG)[16]. Each function call is replaced by an arrow to the subgraph of that function. Outgoing arrows from the subgraph are then reconnected to the graph that contained the function call.

2.5. PATHS IN SOFTWARE

Injection vulnerabilities are characterised by data (or code) entering the program code at a certain point. At a second point, the data leaves the program, triggering the vulnerability. Between these two points, the data travels a path through the program code. The short PHP-program in listing 2.9 illustrates these concepts. Although the data enters the program in line 2, it helps to take the beginning of the path at line 1. Since we know then that *max_count* = 100, we can better reason under what conditions line 4 is reached. It may also be the case that the end point of the path is not an exit point. When analysing a DoS vulnerability, the first line after a WHILE loop could be chosen as the exit point. This is then the first point in the code that is not reached when the program enters an eternal loop.

```
1  $max_count=100;                                //start point
2  $count=$_POST("count");                          //entry point
3  if ($count>$max_count){
4      echo "$count exceeds the max. count ($max_count)"; //exit point
5  };
```

Listing 2.9: PHP code illustrating characteristic points in a code path (start point, entry point and exit point)

Characteristic point	definition
Starting point	point in the code that acts as the starting point of a path.
Entry point	Point in a path where external data enters the path. This is not necessarily the first line. A path may have multiple entry points.
Endpoint	Point in the code that acts as the endpoint of a path. An endpoint may be an exit point. A SQL injection has a path that leads to a point where a SQL query is executed. This is an example of an exit point. Other vulnerabilities, like DoS attack, do have points in code where the malicious input triggers a vulnerability, but without data leaving the program flow. Such a point is an endpoint.
Exit point	Point in the code where data leaves the program. For instance when it is passed as a parameter to an external function.

Table 2.4: Definitions of important points in the path of an injection vulnerability

PATHS IN PHP

In PHP we can identify specific cases where data can enter or leave the code. Examples of such cases are:

- Superglobals¹⁵: `$GLOBALS`, `$_SERVER`, `$_GET`, `$_POST`, `$_FILES`, `$_COOKIE`, `$_SESSION`, `$_REQUEST`, `$_ENV`
- Other predefined variables: `$php_errormsg`, `$HTTP_POST_RAW_DATA`, `$http_response_header`, `$argc`, `$argv`
- Code injection by manipulating `include/readfile/requireonce`
- Return values of functions¹⁶, like `fgets`, `readfile`, `file_getcontents`), `shell_execute`

At the end of the path, data could leave the code in specific places. Examples of such exit points in PHP code are:

- Executing a (No-)SQL query using the `query` command
- Manipulating XML or XPATH commands
- LDAP
- Producing output to a web page using `ECHO`
- Writing data to a file or stream
- Executing a shell command (`shell_exec`)

Larger PHP programs are divided into separate files (modules). Functions and objects can be grouped logically in this way. In PHP there is no standard name for code within a module that is not in a function or object. By analogy with C, we call this code *main*. When starting a script with the main code, functions can be called from this code. In this way, a path is created in which the code flow moves from one module to another. For example, the exit point may be in another module than the entry point (see Figure 2.7). It is possible in PHP to declare inner functions and closures.

If classes are used, a constructor is called when an object is created. This is comparable to a function call. The difference is that in the case of inheritance the constructor of the parent must also be executed. A method call of an object is also similar to a function call. But here too inheritance may occur. Figure 2.8 shows an example of a path that spans several modules in this way.

¹⁵<https://www.php.net/manual/en/reserved.variables.php>

¹⁶<https://www.php.net/manual/en/ref.filesystem.php>

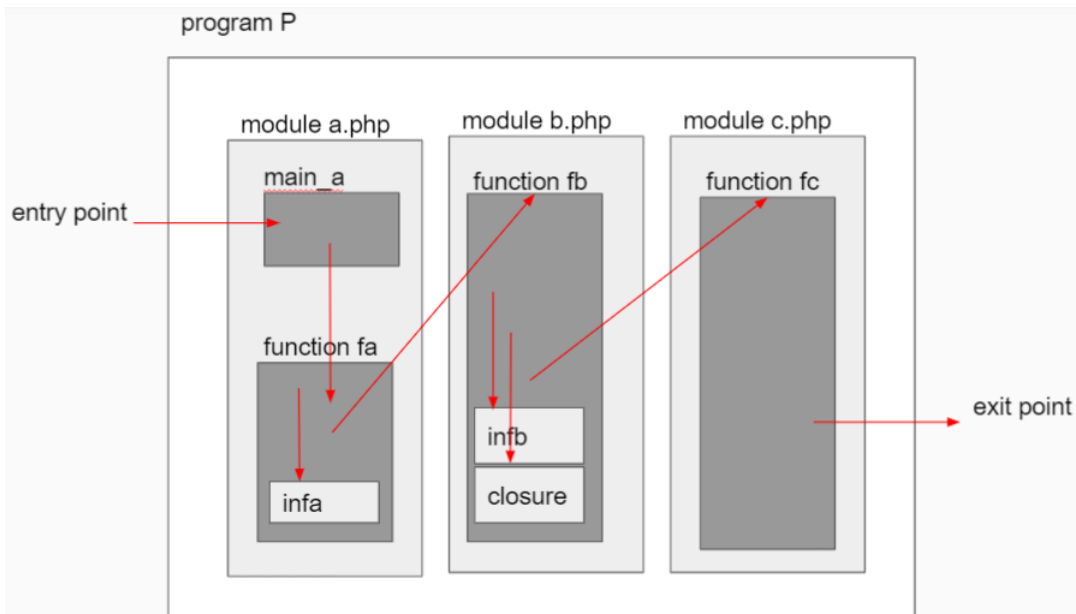


Figure 2.7: Structure of a PHP program

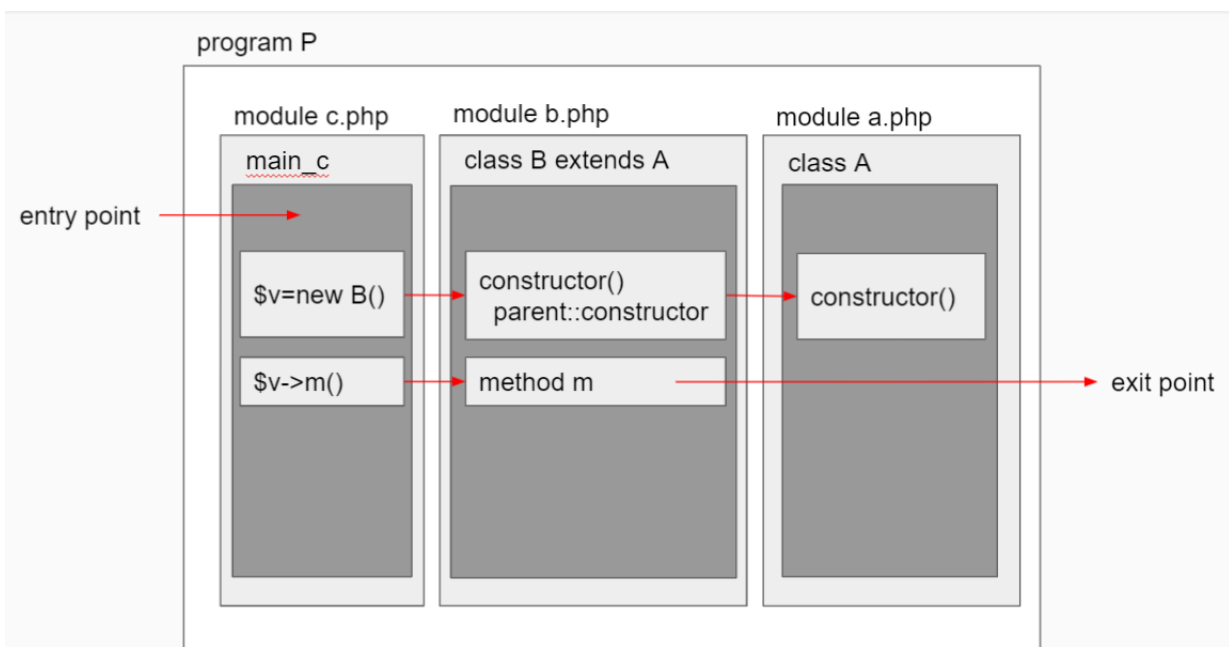


Figure 2.8: Structure of a PHP program with classes

2.6. TRIGGERING PATHS

Paths are only traversed under specific conditions. These conditions are determined by the conditions in the control flow statements (IF, WHILE, DO WHILE, SWITCH, FOR, FORE-ACH) in the path. The set of conditions that determine whether a path can be run through are called path conditions. In the example of listing 2.10, the ELSE branch of the program is chosen if the condition C is not true. Examples for the condition C in this situation are $True$ or $x! = 2$.

```

1  $x=2;
2  if ( C ) {
3      // the condition C is True
4      ...
5  } else {
6      // the negation of the condition C is True
7      ...
8  }

```

Listing 2.10: The condition C determines which path will be chosen

In Figure 2.9 we can determine whether the program can reach line 6 (echo $\$x+\y ;) because after substituting $x = 20$, $y = 40$ in $x < 20 \&\& y > 60$ the condition evaluates False. We conclude that line 6 is not reachable. The same procedure for line 8 (echo $\$x-\y ;) leads to the conclusion that the negation of $x < 20 \&\& y > 60$ evaluates to True.

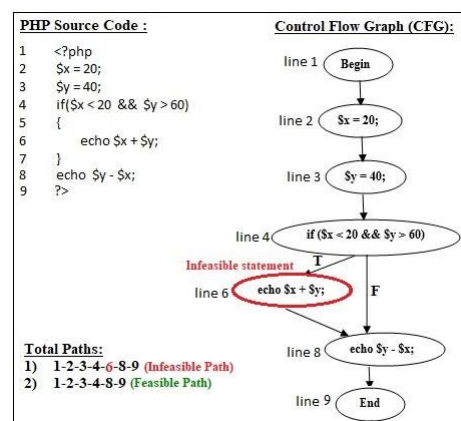


Figure 2.9: Example of a PHP program and the accompanying CFG [35]

A program uses variables. The set of all variables used within a certain context (a module or a whole program) is called P . Some of these variables are used to determine the value of expressions of path conditions. These are not only the variables that actually appear in the expression, but also the variables whose value has been previously assigned to them. The values of these variables determine whether a point in the code can be reached. We call this set R . The values of a third set of variables V are used at the point in the code where the vulnerability is triggered. For example, the value of these variables ensures that a loop never ends, or the value is used in output. The sets R and V are a subset of P : $R \subset P$, $V \subset P$. The intersection $R \cap V$ can be non-empty.

At line 7 in listing 2.11 the sets P , R and V are:

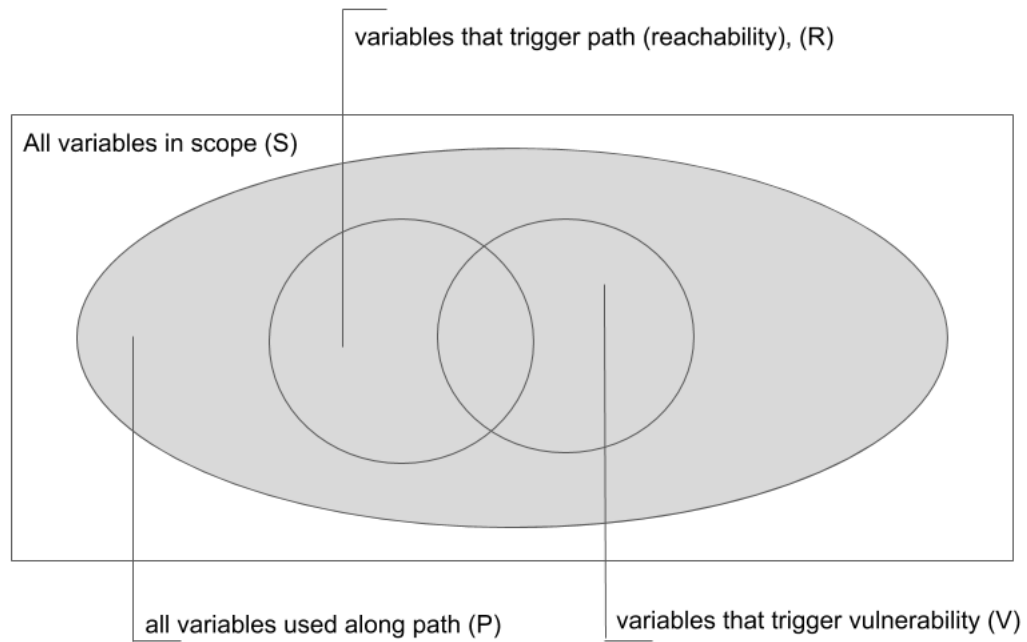


Figure 2.10: Overview of variable collections used in path code. The intersection of the sets R and V can be empty.

- $P = \{ \$a, \$b, \$c, \$username \}$
- $R = \{ \$a, \$b, \$c \}$
- $V = \{ \$username \}$

```

1  $a=$_POST( "a" );
2  $b=$_POST( "b" );
3  $username=$_POST( "username" );
4  $c=$a+$b;
5  if ( $c==3)
6  {
7      echo( $username );
8  };

```

Listing 2.11: example to illustrate the meaning of the variable set P, R and V.

2.7. SYMBOLIC EXECUTION

When a program is executed by a computer, the variables are assigned concrete values. The value can be a constant or it can come from some form of input. This can be input from a human user from an input field, or it can be input from some other external source. With concrete execution, a single path in the program is examined depending on the specific input values. The concrete input values unambiguously determine the value of expressions in control structures and thus establish which path will be chosen.

In the mid-1970s, symbolic execution was introduced by several authors (e.g. King [29]) as a new static software analysis technique. Symbolic, in contrast to concrete execution, does not assume specific input value values, but abstract values for the input. King describes how conditions of linear equations in a program can be solved using Gauss elimination. The solution describes the condition under which a branch is chosen. In symbolic execution, the symbolic input values do not define an execution path, but instead leave open the possibility of analysing which input is needed for each path.

If we want to know whether a program is vulnerable to Injection vulnerabilities, we must in fact find out for which input a path is triggered, which activates the vulnerability. We need to know which input triggers the path. It is not feasible to try this for all possible input values. Even for a simple program, the collection of all possible input values is already so large that it is impossible to analyse them all.

For example, in the example program in 2.12, we can quickly see that line 4 can be reached if $x=2$ and $y=1$. From there, line 7 can be reached if the condition of the IF statement in line 5 is also met.

```
1 $x=$_POST["x"];
2 $y=$_POST["y"];
3 if ($x+2*$y==4)
4 {
5     if (2*$x-$y==0)
6     {
7         echo('valid values');
8     }
9 }
```

Listing 2.12: Example of two nested IF-statements

The number of possible values for x and y is already large even in this simple program. Mathematically, the number of possibilities is infinite, but we can determine for which values of x and y line 7 is achieved, by solving the following system of linear equations.

$$\begin{cases} x + 2y = 4 \\ 2x - y = 0 \end{cases}$$

The mathematical solution to this system is $x = \frac{4}{5}$ and $y = \frac{8}{5}$.

If we restrict x and y to n -bit integers, there are 2^{2n} combinations possible. Firstly, this number is not infinite, but still a large number to check all possibilities one by one. Secondly, we run into a limitation of the method. The fractions $x = \frac{4}{5}$ and $y = \frac{8}{5}$ cannot be represented as a binary number with a finite bit row in the data types that software tools, such as Z3, use to solve this type of condition. Thus, although there is an exact mathematical solution, this is not the solution that determines the conditions under which the system

is solvable if we represent binary numbers with a finite bit row (for example, a byte, integer or word).

Symbolic execution requires keeping track of the following two elements for each path. First, a first-order Boolean formula [27] that defines for each path the conditions under which that path can be executed. Secondly, a collection of variables and their symbolic values.

```

1 function conditionalswap(&$x, &$y) {
2     if ($x>$y) {
3         $x=$x+$y;
4         $y=$x-$y;
5         $x=$x-$y;
6     }
7     assert ($x<=$y);
8 }

```

Listing 2.13: Example of a PHP program that swaps the values of the variables \$x and \$y if \$x>\$y.

In listing 2.13 we see a program in PHP that swaps the value of variables \$x and \$y when \$x>\$y. There are two code paths:

- if \$x<=\$y lines 1 → 2 → 7 are executed.
- if \$x>\$y lines 1 → 2 → 3 → 4 → 5 → 7 are executed.

After executing the function, the condition \$x<=\$y must always hold. We can check for concrete values of \$x and \$y for which this is correct (take for example \$x=7 and \$y=1), but in general we want to know for all possible values of \$x and \$y that the function swaps the values of the variables \$x and \$y if necessary. We can do that by taking the symbolic values A and B for the function parameters \$x and \$y.

$\$x \rightarrow A$
 $\$y \rightarrow B$

The expression \$x>\$y in line 2 has the symbolic value $A > B$. If $A > B$ is not true, lines 3, 4 and 5 are skipped. In other words, this path is only executed if $\neg(A > B)$. Since in this case no more statements are executed after the IF statement, after going through this path it is still true that $\neg(A > B) \equiv A \leq B$.

If $A > B$ holds in line 2, lines 3, 4 and 5 are executed.

after line 3:	$\$x = \$x + \$y;$	$\$x \rightarrow A + B$
after line 4:	$\$y = \$x - \$y;$	$\$y \rightarrow (A + B) - B = A$
after line 5:	$\$x = \$x - \$y;$	$\$x \rightarrow (A + B) - A = B$

These lines effectively swap the values of the variables \$x and \$y, so after line 5 the condition \$x<=\$y does apply. So in both cases ($\$x \leq \$y \wedge \$x > \y) the function ends with $\$x < \y : the condition ($\$x < \y) is met in all cases. We can see from this example that the conditions along a path define the logical formula that determines the conditions under

which that path is run. The assignments along a path determine the (symbolic) values of the variables.

The program that executes the symbolic execution is called a symbolic execution engine. When executing a statement, the symbolic execution engine keeps track of a state ($stmt, \sigma, \pi$) [5] [27]. These three elements stand for:

- $stmt$ is the next statement in the path.
- The symbolic store σ is the collection of variables with their (symbolic) value.
- path conditions π is the logical formula that determines under what conditions the current point in the path can be reached (the collection of conditions that were true).
At the start of the program we take $\pi = true$ as the initial path condition.

The statement $stmt$ determines how the current condition changes. With an assignment $x = e$, the symbolic store σ is modified so that the value of the variable x becomes equal to the value of the expression e that follows from the current values in σ . If the statement is an IF statement $IF(e)$, the value of the boolean expression e determines which path will be traversed. If $e == true$, the THEN path is executed. Therefore, we add the condition e to the path condition π of the THEN-path. The new path conditions will then be $\pi \wedge e$. To the ELSE path we add the condition $\neg e$. The path condition of the ELSE path then becomes $\pi \wedge \neg e$.

The Symbolic Execution Tree in figure 2.11 shows the state after each statement.

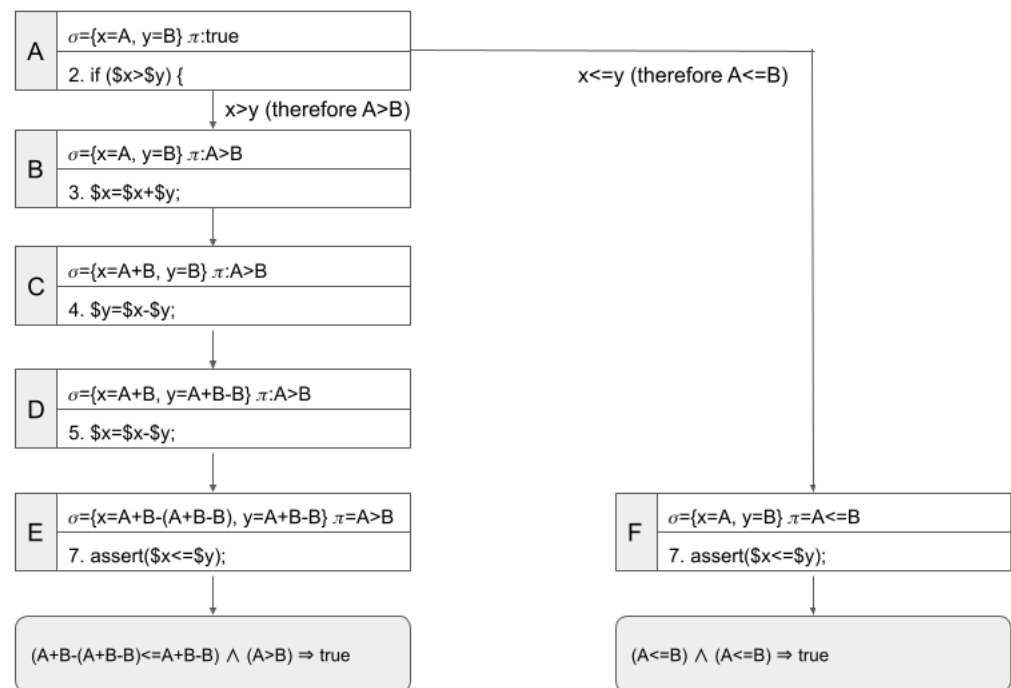


Figure 2.11: Symbolic execution tree

If we want to use symbolic execution when analysing source code, we run into a number of bottlenecks [5].

- *State explosion.* A concrete path through a program is called a path. For each choice, there are (at least) two paths to choose from. For n consecutive choices between m possibilities, the number of paths is m^n . The number of paths thus increases exponentially. Symbolic Execution may be an approach that does not consider one single concrete path, but (in theory) all paths simultaneously. Actually evaluating all paths quickly becomes impossible due to the large number of possibilities.
- *Memory.* To simple variables we can assign a symbolic value. It gets more complicated with pointers (not present in PHP), arrays or objects.
- *The environment.* When executing a program, we soon come into contact with the environment of the program [11]. The environment can consist of external files (that the program creates or edits), the execution of system libraries, but also functions in the programming language itself that need to be evaluated.
- *Evaluation path constraints.* To what extent can we resolve conditions? For example, we might be able to solve some conditions manually, but to what extent is an automated tool able to do so?

Fuzzing [22] and symbolic execution are in a sense dual test methods.

fuzzing	symbolic execution
The test method selects inputs from a set of all possible input values (for the variables that input directly into).	The test method selects paths from the set of all possible paths through the program (for the paths between entry point and exit point).
Selection of all possible sets of input values (limited by type). Each set activates a single path (sometimes no full path is triggered).	Selection from the set of all possible paths (limited by search depth or another (random) criterion). If a path is traversable, there is a set of concrete input values that we can fill in for all (symbolic) variables that appear in the path conditions for this path, so that all conditions are true.

Table 2.5: fuzzing versus symbolic execution

2.8. STATIC SINGLE ASSIGNMENT (SSA)

Static Single Assignment is a notation form devised to simplify and improve compiler optimisations[1]. SSA means that each variable is assigned a value only once. This can be done in a simple way by replacing a variable x with x_i , where i is a counter that keeps track of which version of the variable it is. In standard code, the assignment $x = y$ does not mean that we can replace all instances of x by y , because the value of x may be unequal to y at some other point in the code due to other assignments. If in SSA form the assignment $x_i = y_i$ occurs, we can replace all instances of x_i by y_i . We thus save the variable x_i .

If the code has no branches or loops, a *basic block*, the conversion of code to the SSA variant is straightforward. Each assignment introduces a new version of the variable. The counter becomes 1 in the case of a new variable. In the case of a new assignment to an existing variable, the counter is incremented.

standard code	SSA form
$x = 1$	$x_1 = 1$
$y = 2$	$y_1 = 2$
$x = x + y$	$x_2 = x_1 + y_1$
$y = x * x$	$y_2 = x_2 * x_2$

Table 2.6: standard code of a basic block converted to SSA form

2.8.1. BRANCHES

After branching, we need to know which version of the variables to work with. To do this, we introduce the ϕ -function (See figure 2.12). The ϕ -function is inserted at the beginning of a *basic block* with more than one incoming arrow. If several versions of a variable are received, the ϕ -function determines which version should be used. Which version that is, depends on the actual path that was taken.

The code in table 2.7 is visualised as a CFG with statements in SSA form in figure 2.12.

standard code	SSA form
$x=6;$	$x_1=6;$
if odd(x) {	if odd(x_1) {
$x = 3 * x + 1;$	$x_2 = 3 * x_1 + 1;$
} else {	} else {
$x = x/2$	$x_3 = x_1/2;$
}	}
print(x);	print($\phi(x_2, x_3)$);

Table 2.7: standard code of a branch converted to SSA form

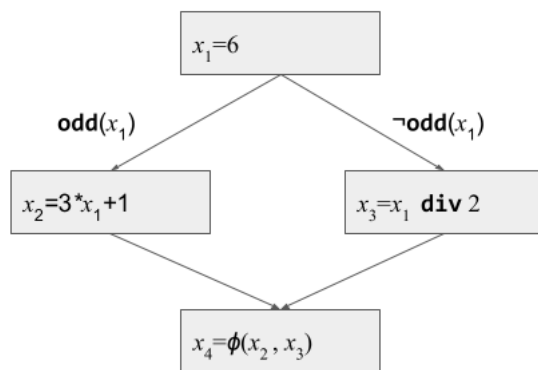


Figure 2.12: Example of the use of the ϕ function in a basic block after a branch.

2.8.2. LOOPS

A similar analysis applies to a loop. We illustrate this using a while loop. The while statement becomes a block with two incoming arrows: one from the previous block and one two from the block in the while loop. A ϕ function is used to determine which version of a variable is needed. The example code in table 2.8 is visualised in figure 2.13.

standard code	SSA form
$x=6$	$x_1=6$
	label1: $x_2 = \phi(x_1, x_5)$
while $x>1$	if $x_2 > 1$
{	{
print(x)	print(x_2)
if odd(x)	if odd(x_2)
{	{
$x = 3 * x + 1$	$x_3 = 3 * x_2 + 1$
} else {	} else {
$x = x/2$	$x_4 = x_2/2$
	}
	$x_5 = \phi(x_3, x_4)$
}	goto label1
}	}
print(x)	print(x_2)

Table 2.8: standard code of a while loop converted to SSA form

There are several known algorithms for converting code to SSA form. A first algorithm comes from Appel [3], which was later improved significantly (Aycock et al. [4], Cooper et al. [13], Braun et al. [7]).

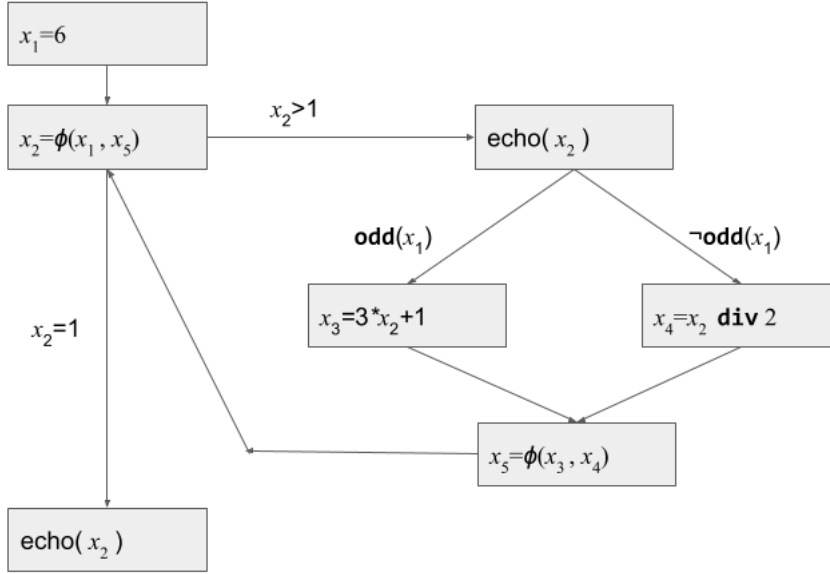


Figure 2.13: Example of the use of the ϕ function in while loop, which contains a branch

2.9. SMT SOLVERS

We have seen in the previous subsections that the solutions of the path condition are the conditions that determine whether a path is traversable. But how do we determine these solutions?

In King's original article, conditions of linear formulas are solved using Gauss elimination. With this method, only solutions to problems that can be formulated as a system of linear equations can be calculated. A more general approach has been developed starting from the problem of finding solutions to Boolean formulas (the Boolean satisfiability problem, abbreviated as SAT). This generalization is known as the satisfiability modulo theories problem (abbreviated as SMT). In addition to Boolean formulas, more complex conditions involving real numbers, integers, and/or various data structures such as lists, arrays, bit vectors, and strings can be used. The programs that can solve these mathematical problems are called SMT solvers.

examples

The formula $x_1 \wedge x_2$ becomes true only for $x_1 = \text{True}, x_2 = \text{True}$. The formula $x_1 \vee x_2$ can be made true in three ways, for example by taking $x_1 = \text{True}, x_2 = \text{False}$. Both these logical formulas are therefore satisfiable.

The formula $x_1 \wedge \neg x_1$ is unsatisfiable, because it can not be made true by choosing a value for x_1 in any way.

Despite the fact that Cook [12] proved that the SAT problem is NP-complete, there is a highly efficient algorithm for propositional formulas in conjunctive normal form. The DPLL algorithm uses backtracking to find assignments satisfying the conditions in the search space [15], [14]. Over the years, research has shown that efficient improvements in the algorithm and its implementation are possible [40].

The main reason that SMT solvers with the DPLL algorithm are useful for solving path constraints is that they support extensions, called Theories. There are Theories for Linear

Algebra, Non-Linear Algebra, Real numbers, Strings and Arrays [31].

Each extension maps the problem to a CNF-SAT problem. If this formula is feasible, the specific algorithm for the theory examines whether the extension is also feasible. If so, the problem is sat.

Because SMT solvers can determine solutions for conditions in more situations (theories), there are, at least in theory, more opportunities to solve path constraints. To what extent this is actually possible is the subject of our research.

SMT solvers can be addressed in two ways: via an API or via a standardised language (SMTLIB¹⁷). The advantage of SMTLIB is that scripts in this language are understood by many SMT solvers. This makes it possible to exchange SMT solvers. Furthermore, the SMTLIB language is human readable, which makes it easy to check or modify the input for the SMT solver.

¹⁷<https://smtlib.cs.uiowa.edu/language.shtml>

3

RESEARCH DESIGN

3.1. RESEARCH QUESTIONS

In this chapter, we explain the method by which we intend to find an answer to the research questions we formulated in response to the main question.

The main question of our research is: *"How to determine input values that trigger paths to injection vulnerabilities in PHP-code using Symbolic Execution?"*. To answer this main question, we have divided it into three research questions.

- RQ1: How can we find paths in PHP-code efficiently?
- RQ2: Up to what extent can PHP-code be evaluated using Symbolic Execution?
- RQ3: Up to what extent can path conditions be solved by an SMT solver?

Our research is connected with previous research in several aspects. For the search of a path between two nodes in a graph, several algorithms are known [44]. The limits of the use of symbolic execution have been studied previously by previous research does not specifically mention to what extent the use of a certain SMT solver is a limitation for solving path conditions for programs written in PHP.

In particular, our research is consistent with previous research done at the Open University Netherlands. Beisicht [6] investigated how Injection Attacks can be mitigated by using Secure Multi Execution [17]. Kronjee [32] studied the extent to which dataflow analysis and machine learning can detect vulnerabilities in PHP programs. Elema [19] investigated whether Deep Learning on graph representations can detect vulnerabilities.

3.2. RESEARCH METHOD

In this section, we describe the research method we will use for each research question. All digital artifacts resulting from the research can be found on github ¹. In addition smaller tools were used, for example, for the preparation of the datasets. Also for all these tools the code and the instructions for use can be found on github.

RQ1: HOW CAN WE FIND PATHS IN PHP-CODE EFFICIENTLY?

The source code of a PHP program consists of text. The structure of this program code can be determined in an automatic way with a parser. The output is an abstract syntax tree (AST). With the help of this AST we can determine the structure of the program. The program structure includes variables, expressions, statements and functions. Conditional statements are of particular importance because they determine the paths along which the program can be run. CFGs are a common way to represent the paths in a program as a graph. In a CFG (graph) we can search for paths between two nodes, by looking for a connection through intermediate nodes. There are standard algorithms known that can be used to search a simple paths between two nodes (Breath First Search, Depth First Search [44]). For our purpose, these algorithms are not suitable, because we want to have influence on certain properties of the paths. For example, we want to be able to control how often a cycles a path, because a code path that triggers a vulnerability is not necessarily the shortest path. We can summarize the sub-steps as follows:

- How can we obtain an AST of a PHP program? Furthermore, we investigate which software we can use for software development.
- From an AST (a tree structure), we want to describe how it can be incorporated into a CFG. We choose a CFG because it is a suitable structure for searching paths.
- We want to cover as much of the language elements of PHP as possible. Therefore, we will build a test set that allows us to use language elements isolated or combined to create ASTs.
- Finally, we want to determine how we can efficiently search paths in a CFG?

RQ2: UP TO WHAT EXTENT CAN PHP-CODE BE EVALUATED USING SYMBOLIC EXECUTION?

The result of RQ1 is that we have a method by which paths can be searched. RQ2 aims to determine to what extent the statements in a path can be executed by an SMT solver. We will go through the following sub-steps for this purpose. First determine which SMT solvers qualify. We will make a choice based on criteria. Next we will study how to convert statements from a PHP program to conditions for an SMT solver. In doing so, we will determine what possibilities and limitations we encounter. PHP is a dynamically typed language. Therefore, the type of a variable is not fixed in advance and can change with each use. SMT solvers, on the other hand, expect the variables used in conditions to be of a pre-declared type. We must therefore examine to what extent it is possible to correctly declare the type when translating PHP statements into conditions for an SMT solver.

¹<https://github.com/rdohmen/pvpc>

RQ3: UP TO WHAT EXTENT CAN PATH CONDITIONS BE SOLVED BY AN SMT SOLVER?

For RQ3, we want to determine the extent to which the method/tool that follows from RQ1 and RQ2 is actually applicable to PHP code containing injection vulnerabilities. In doing so, we want to make use of the SAMATE dataset. To our best knowledge, there is no other comparable test set available. The Samate dataset consists of a large number of code fragments of which we want to determine the extent to which they can be executed according to our method with symbolic execution, with the goal of finding input that triggers paths to a vulnerability. To do this, we will analyze for each sample which language constructs a tool must master in order to execute this sample symbolically.

OVERVIEW

The different sub-steps of the research are visually represented in figure 3.1.

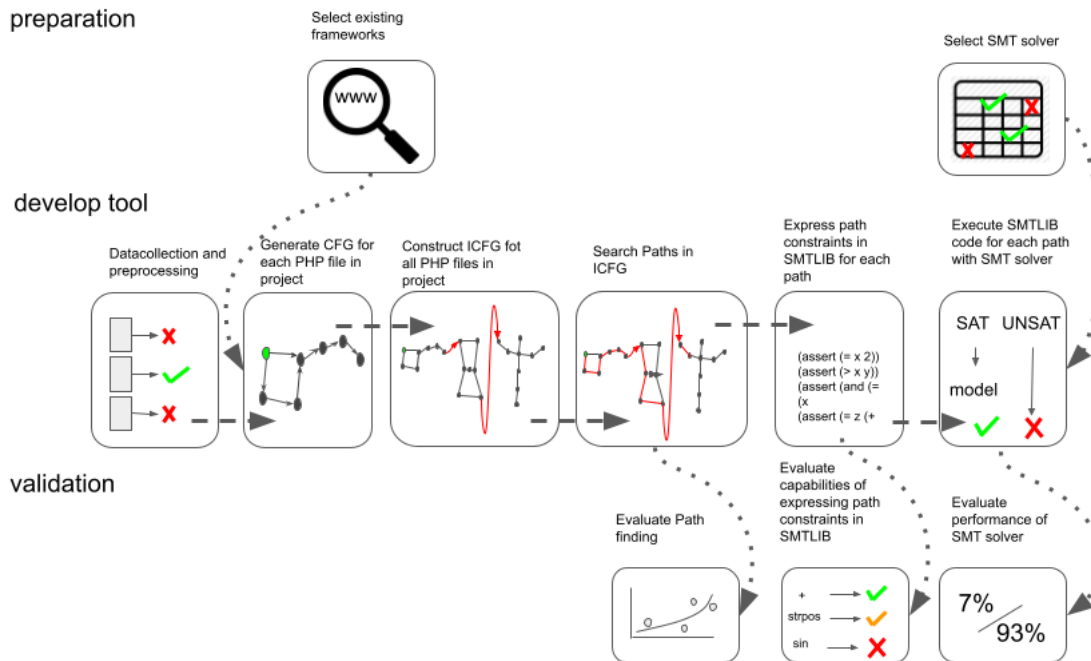


Figure 3.1: Global overview of the stages of our research

The research consists of three phases. In the preparation phase, we analyze existing software frameworks and software tools that may be useful in conducting the research. The results of the preparation phase can be found in Section 3.3. In the second phase, we develop a methodology and software tool with which we aim to answer the three research questions on practical aspects. The results of this can be found in sections 4, 5 and 6. In the final phase, we validate the results of the research.

3.3. SELECTION OF FRAMEWORKS AND SOFTWARE TOOLS

In this section, we describe analyses we conducted for choosing software tools suitable for the study. For the software, we looked at the steps that hneed to be performed. For each

step, we looked at the existing frameworks or components that can be used in a step. Of the possible solutions, we argued which is the most appropriate.

- Step 1: Obtain the AST from PHP source code. A larger project consists of several PHP files. In this case, an AST must be created for each PHP file.
- Step 2: Convert the ASTs to a structure in which code paths can be searched.
- Step 3: Search for code paths between an entry and exit point. Collect path conditions accordingly.
- Step 4: Translate the path conditions into the SMTLIB language.
- Step 5: Using an SMT solver, determine if the path condition is satisfiable. If so, try to obtain a model (a solution).

For each step, we looked at the possibilities of realising this sub-step. The selection process can be visualised as a matrix. The steps are listed horizontally. Possible choices for each step are shown vertically. A chosen option can be indicated by colouring the circles for the chosen option (Figure 3.2).

	step 1 PHP→AST	step 2 Search path	step 3 PC (+ SC)	step 4 PC→SMTLIB	step 5 Solve PC
PHPLY	<input type="radio"/>	PHP <input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Z4 <input type="radio"/>
PHPAST	<input type="radio"/>	LAZARUS <input type="radio"/>	<input type="radio"/>	<input type="radio"/>	CVC4 <input type="radio"/>
PHPSCAN	<input type="radio"/>	PYTHON <input type="radio"/>	<input type="radio"/>	<input type="radio"/>	MathSat <input type="radio"/>

Figure 3.2: Design matrix for the tool

Apart from the implementation choices of the tool, there is also the question of which PHP version the tool will support. There are minor differences in syntax between different versions of PHP. Moreover, there are differences in statement execution between different versions, even if the exact same code is used². We could have chosen to make the operation of the tool depend on a PHP version number in the relevant places, but we did not dare to include this complicating factor for time reasons.

3.3.1. ANALYSIS OF EXISTING TOOLS FOR CREATING AST'S

We compared a number of existing components to assess their applicability to the study.

- PHPLY³ was used in scientific research before [32].
- Phpscan⁴ has not been developed for several years and will therefore not be a future-proof choice.

²<https://www.php.net/manual/en/language.operators.comparison.php>

³<https://github.com/viraptor/phply>

⁴<https://github.com/bartvanarnhem/phpscan>

- Php-ast ⁵, further development of php-parser ⁶. Php-ast is being actively developed by a developer who is also involved in the development of the PHP interpreter. The module uses the internal AST introduced earlier in PHP7.
- Phpast ⁷, tool of the same name that has not yet been developed very far.
- Peachpie ⁸ was previously used in research by [6]. Possible disadvantage could be that we make the tool too dependent on too many external tools.
- Rascal is a meta-programming language that, among other things, has the ability to analyse PHP code [25]. Although Rascal is being developed reasonably actively ⁹, the software is not as stable as other development environments.
- CHEF's [8] approach of adapting the interpreter to enable Symbolic Execution may be extended to include the possibility of using the AST in the interpreter to achieve other steps in the tool's process. The disadvantage of this approach is that modifications are bound to a certain version of the interpreter.

Another consideration is to write your own PHP parser. We looked at the possibilities of building a parser with ANTLR ¹⁰ or Lex/Yacc ¹¹. For Antler, there is a PHP grammar, which makes it possible to generate a parser quickly. We succeeded in doing so, but due to the limited time available, it was not considered worthwhile to build on this. With Yacc/Lex, it was not possible to generate a basic parser within a reasonable time. It was also questionable whether the grammars that can be found in various places would be useful. Based on these considerations, we looked further into PHPLY and PHP-AST. Both proved to have good support for the various AST nodes needed to analyse language structures in PHP. An overview of the similarities and differences can be found in the table in Appendix A.

The decisive reason for choosing PHP-ast was that the generated AST files are easy to interpret. In addition, the developer is also one of the developers of PHP, which gives confidence in good support.

3.3.2. CHOOSING APPROPRIATE PROGRAMMING LANGUAGES

These steps involve the choice of a programming language. The main consideration is whether it is possible to program the required algorithms. Kronjee [32] has done this before in Python. PHP would fit well with the choice of PHP-AST (steps 1 to 4 can then be programmed in PHP) or PHPLY (the same steps can then be programmed in Python). The main reason for not using these languages is that the programming language Lazarus ¹² (Object Pascal) is a more familiar language to us.

For the actual development we used the following tools.

- PHP, version 7.4.13 ¹³ There is no reason to believe that a more recent version of 7.4.x

⁵<https://github.com/nikic/php-ast>

⁶<https://github.com/nikic/PHP-Parser>

⁷<https://github.com/flaviovs/phpast>

⁸<https://www.peachpie.io/>

⁹<https://update.rascal-mpl.org/>

¹⁰<https://wwwantlr.org/>

¹¹<http://dinosaur.compilertools.net/>

¹²<https://www.lazarus-ide.org/index.php?page=downloads>

¹³<https://windows.php.net/download/>

would not work. Version 8 does have changes that affect how the tool works. The same is true for older versions (<7.4.x).

- Lazarus 2.0.6, FPC 3.0.4 ¹⁴
- Python 3.8.2 ¹⁵

3.3.3. CHOOSING AN APPROPRIATE SMT SOLVER

In the last step, the path conditions must be solved by an SMT solver. Two factors must be taken into account here. The choice of a SMT solver and the link between the SMT solver and our tool. The latter factor depends on the programming language. We have not found an API coupling between PHP and a modern SMT solver. For Python, several sources to SMT solvers can be found ^{16 17}. For Lazarus, no coupling existed, but there is a coupling with Delphi (Z34Delphi ¹⁸), which might be usable in Lazarus too.

Over time, many different SMT solvers have been developed. To determine which SMT solver is suitable for our research, we looked at a number of aspects. If we want to make the choice somewhat future-proof, we could look at whether the solver is still being actively developed. The solver should not only determine whether the problem is satisfiable (sat), but the solver should also be able to provide a model, because we are looking for concrete examples of input that can trigger a path. Thus, we not only want to know whether the conditions for a path are satisfiable, but also (at least) get an example of such an input. Furthermore, the solver must be able to solve problems within a number of theories. Simple conditions like $x < 0$ are covered by linear algebra.

This kind of condition could be solved, as in King's original article, with Gauss elimination, i.e. without a SMT solver. However, one of the extensions that are included in modern SMT solvers is the Theory of Linear Algebra. Thus, with an SMT solver we can use propositional conditions, but also linear conditions. The same is true for non-linear problems. Although it is known that SMT solvers within this Theory are incomplete. As a consequence, SMT solvers for trivial problems sometimes not only do not give a solution, but judge the problem is unsatisfiable. An example is shown in Listing 3.1¹⁹.

```
1 (declare-const x Real)
2 (declare-const y Real)
3 (declare-const z Real)
4 (assert (= (* x x) (+ x 2.0)))
5 (assert (= (* x y) x))
6 (assert (= (* (- y 1.0) z) 1.0))
7 (check-sat)
```

Listing 3.1: trivial nonlinear problem rated as unsat by Z3 (the solution is $x=0$, $y=2$, $z=1/2$)

In terms of data types, integers are a reasonable first entry, but in a computer, integer numbers are modelled as bit vectors with a certain length. In addition, data of type

¹⁴<https://www.lazarus-ide.org/index.php?page=downloads>

¹⁵<https://www.python.org/downloads/release/python-382/>

¹⁶<https://pypi.org/project/PySMT/>

¹⁷<https://github.com/Z3Prover/z3>

¹⁸<https://github.com/Pigrecos/Z34Delphi>

¹⁹<https://rise4fun.com/Z3/tutorial/guide>

string occurs frequently in injection vulnerabilities, for example, when data is entered via a `$_POST` command. Therefore, it is important that the SMT solver can operate with string conditions. Therefore, we prefer a SMT solver that can evaluate conditions with bit vectors and strings. If we want to keep the software independent from a certain SMT solver, so that it can be exchanged for another solver in the future, we would preferably like to use the SMTLIB language. The final wish is that the solver can be addressed via an API from a modern programming language.

	last update	unsat	model	LIA	NLIA	A(X)	BV	Real	String	SMTLIB	API
CVC4 ²⁰	may 2021	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	C++
Hampi	unknown	No	No	No	No	No	No	No	Yes	No	Java
MathSAT ²¹	april 2021	Yes	Yes	Yes	No	Yes	Yes	Yes	No	Yes	C
VeriT ²²	october 2016	No	Yes	Yes	No	No	No	Yes	No	Yes	C
Yices ²³	april 2020	No	Yes	Yes	No	Yes	Yes	Yes	No	Yes	C
Z3 ²⁴	june 2021	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	C, C++, .NET, Java, OCaml, Python

Table 3.1: Characteristics of modern SMT solvers

Our conclusion based on the Master thesis of Andres Höfler²⁵ and our own research is shown in Table 3.1. In summary, the following two possibilities seemed the most obvious, looking at all factors.

- option 1: Writing the whole tool in Python, using PHPLY for the AST, and Z3 as SMT solver.
- option 2: For step 1, use PHPAST. Writing steps 2 to 5 in Lazarus, assuming that the Delphi link to Z3 is convertible to Lazarus.

For the first option, it speaks for itself that the whole tool can be written in one language using existing components that work reliable. The programming in Python is however moderate compared to other languages. For the second tool, the information present in the AST is excellent. The experience in programming in Lazarus is greater than with other programming languages, which gives more confidence of success.

In the end, there was no deciding factor, so option 2 was chosen more or less at random (See figure 3.3).

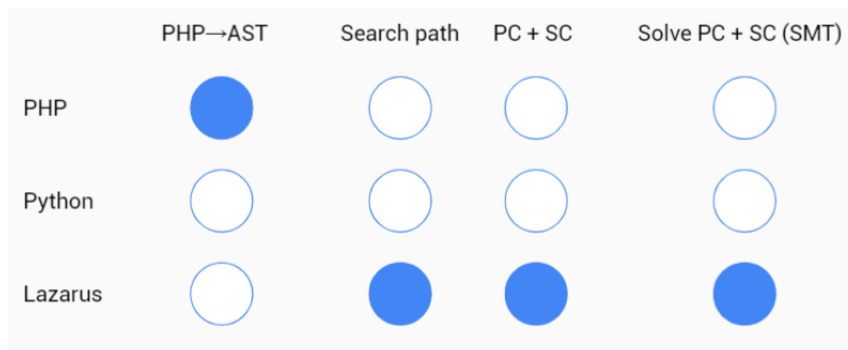


Figure 3.3: Initial design of the tool (PC: Path Conditions, SC: Security Conditions)

Because at the end it turned out that the link with the SMT solver did not work properly from Lazarus, we finally had to choose for a link from Python. This made it impossible to

²⁵https://spreadsheets.ist.tugraz.at/wp-content/uploads/sites/3/2015/06/DS_Hoefler.pdf

write a monolithic tool (without starting from scratch again). This led to the design in figure 3.4.

	PHP→AST	Search path	PC + SC	Solve PC + SC (SMT)
PHP	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Python	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
Lazarus	<input type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>

Figure 3.4: Final (adapted) design of the tool

3.4. TESTING

We tried to ensure code quality by writing unit tests for some components of the main tool to test the operation of the isolated component. The source code for this can be found on the aforementioned github page.

3.5. DATA DESCRIPTION

3.5.1. TEST FILES WITH ISOLATED PHP CODE

To test language elements of PHP in isolation or combined, we created a test set with a series of PHP files. Because we also wanted to test larger projects that consist of more than one source code file, we have created small test projects in a similar way²⁶.

3.5.2. PHP VULNERABILITY TEST SUITE

For the study, we make use of the PHP Vulnerability Test Suite²⁷ by Bertrand C. Stivalet and Aurelien Delaitre which can be found on the NIST website²⁸. This dataset provides thousands of PHP code snippets for various CWEs that have been classified as safe or unsafe (see table 3.2).

The source code of the files shows whether a sample is classified as (un)safe. In listing 3.2, an example can be seen of one of the files²⁹.

```
1 <?php
2 /*
3  Safe sample
4  input : backticks interpretation, reading the file /tmp/tainted.txt
5  SANITIZE : use in_array to check if $tainted is in the white list
6  construction : interpretation with simple quote
7  */
```

Listing 3.2: Example of a marking of the sample type

For unsafe samples, the xml file manifest-103-RAjiQO.xml contains the line number where the vulnerability can be found. We have written a tool that splits the files up according to CWE and adds the line number where the vulnerability manifests itself to the relevant PHP file, so that this information can be read immediately in the PHP file.

²⁶<https://github.com/rdohmen/php-test-set>

²⁷<https://samate.nist.gov/SARD/view.php?tsID=103>

²⁸<https://www.nist.gov/>

²⁹CWE_98__backticks__whitelist_using_array__require_file_name-interpretation_simple_quote.php

CWE number	Number of safe samples	Number of unsafe samples
078	1872	624
079	5728	4352
089	8640	913
090	1728	2112
091	4785	1263
095	1296	336
098	2592	672
209	5	3
311	2	2
327	3	5
601	2208	2592
862	400	80

Table 3.2: Number of safe and unsafe samples for all CWE categories present in the SAMATE testset.

4

FINDING PATHS

4.1. RQ1: HOW CAN WE FIND PATHS IN PHP-CODE EFFICIENTLY?

We want to determine input values that trigger a path. The entry point and the exit point of the path are known. Our tool must determine which code path is run between these two points. It is possible that there are several paths between entry and exit points. A conditional statement like an IF or SWITCH statement can result in multiple paths between entry point and exit point that only differ in the branches of the IF/SWITCH statement. A WHILE statement can cause a path to differ from another path because the number of times the loop is run differs. In a similar way, this can also happen with DO or FOR statements. When paths are found, the conditions and statement for this path must be translated into conditions for the SMT solver. It is then up to the SMT solver to determine whether the conditions are satisfiable and whether a model for the conditions can be found.

4.2. CREATE AST'S FROM PHP-FILES

In order to get an AST we use a PHP program: `astarg.php`. This is a small PHP program that uses an `ast_dump`-function that is part of `PHP-ast` (See listing 4.1).

```
1  <?php
2  require './util.php';
3
4  $filename = './'. $argv[1];
5  $code = file_get_contents($filename);
6  $ast = ast_dump(ast\parse_code($code, $version=70),1);
7  $path_parts = pathinfo($filename);
8  // filename is only since PHP 5.2.0
9  \ $astfilename= \ $path_parts[ 'filename' ]. '.ast' ;
10 file\put\_contents($astfilename, $ast)
11 ?>
```

Listing 4.1: source code of `astarg.php`

The program takes a PHP file as an argument and exports a text file with the same name and extension .ast. Running

```
php phparg.php example.php
```

creates a file example.ast.

```
1 <?php
2 $a=2;
3 $b=3;
4 $c=(5-$a)*($b+7);
5 $d=$c/max(2,3);
6 echo(" $c $d");
7 ?>
```

Listing 4.2: source code of example.php

The PHP code of the program in listing 4.2 is thus exported to the code in listing 4.3 representing the AST.

```

1  AST_STMT_LIST @ 1
2      0: AST_ASSIGN @ 3
3          var: AST_VAR @ 3
4              name: "a"
5          expr: 2
6      1: AST_ASSIGN @ 4
7          var: AST_VAR @ 4
8              name: "b"
9          expr: 3
10     2: AST_ASSIGN @ 5
11         var: AST_VAR @ 5
12             name: "c"
13         expr: AST_BINARY_OP @ 5
14             flags: BINARY_MUL (3)
15             left: AST_BINARY_OP @ 5
16                 flags: BINARY_SUB (2)
17                 left: 5
18                 right: AST_VAR @ 5
19                     name: "a"
20             right: AST_BINARY_OP @ 5
21                 flags: BINARY_ADD (1)
22                 left: AST_VAR @ 5
23                     name: "b"
24                 right: 7
25     3: AST_ASSIGN @ 6
26         var: AST_VAR @ 6
27             name: "d"
28         expr: AST_BINARY_OP @ 6
29             flags: BINARY_DIV (4)
30             left: AST_VAR @ 6
31                 name: "c"
32             right: AST_CALL @ 6
33                 expr: AST_NAME @ 6
34                     flags: NAME_NOT_FQ (1)
35                     name: "max"
36                 args: AST_ARG_LIST @ 6
37                     0: 2
38                     1: 3

```

Listing 4.3: AST of listing 4.2

The AST file contains nodes from the AST tree. These nodes have a descriptive name starting with AST_. In listing 4.3 we recognise a number of these nodes. If a node is a child of a previous node, the line is indented with 4 extra spaces. A node on a line with 8 spaces is a child of a node on the first preceding line with 4 spaces. An AST node is followed by an @-sign and a line number (or line numbers). Each child consists of a key, value pair. The children of AST_STMT_LIST have the successive keys 0, 1, 2, ... followed by an AST node

describing the type of statement.

```
AST\_STMT\_LIST @ 1      list of statements
  0: AST\_ASSIGN @ 3 assignment
```

The assignment on line 4 can be found in the tree as 1: AST_ASSIGN @ 4. This is the second statement in the statement list. The node of statement 4 again has two children.

```
1: AST\_ASSIGN @ 4
  var: AST\_VAR @ 4
    name: "b"
  expr: 3
```

The var key has the value AST_VAR and indicating it describes a variable. The name is described in the second child of the node: the "name" (key) is "b" (value).

```
name: "b"
```

The expression on the right hand side of the assignment is simple in this case

```
expr: 3
```

The expression consists of the value 3. The third statement has a somewhat more complicated expression. The expression contains a binary operator. The child node determines that it is a multiplication operator.

```
expr: AST\_BINARY\_OP @ 5
  flags: BINARY\_MUL (3)
```

The two operands of the multiplication are worked out in the nodes with the keys left and right. The left node is a binary operator. In this case, a subtraction. The two operands are 5 and the variable a.

```
left: AST\_BINARY\_OP @ 5
  flags: BINARY\_SUB (2)
  left: 5
  right: AST\_VAR @ 5
    name: "a"
```

```
2: AST\_ASSIGN @ 5
  var: AST\_VAR @ 5
    name: "c"
  expr: AST\_BINARY\_OP @ 5
    flags: BINARY\_MUL (3)
    left: AST\_BINARY\_OP @ 5
      flags: BINARY\_SUB (2)
      left: 5
      right: AST\_VAR @ 5
        name: "a"
```



```

right: AST_BINARY_OP @ 5
  flags: BINARY_ADD (1)
  left: AST_VAR @ 5
    name: "b"
  right: 7

```

In the fourth statement, we encounter a function call.

```

right: AST_CALL @ 6
  expr: AST_NAME @ 6
    flags: NAME_NOT_FQ (1)
    name: "max"
  args: AST_ARG_LIST @ 6
    0: 2
    1: 3

```

The name of the function is max. This function takes arguments 2 and 3.

Our program reads in an AST file and processes it into an internal representation of this AST. It starts with a root node connected to AST nodes which have a key and value. Below this are possibly child nodes.

4.3. MORE ABOUT ASTs

An AST is generated from each PHP source code file. An AST may contain function or class declarations (See Figure 4.1). These declarations form a subtree in an AST (See Figure 4.2). Functions or methods can be called from the main code or from other functions and methods. Moreover, functions and classes from another file can be used.

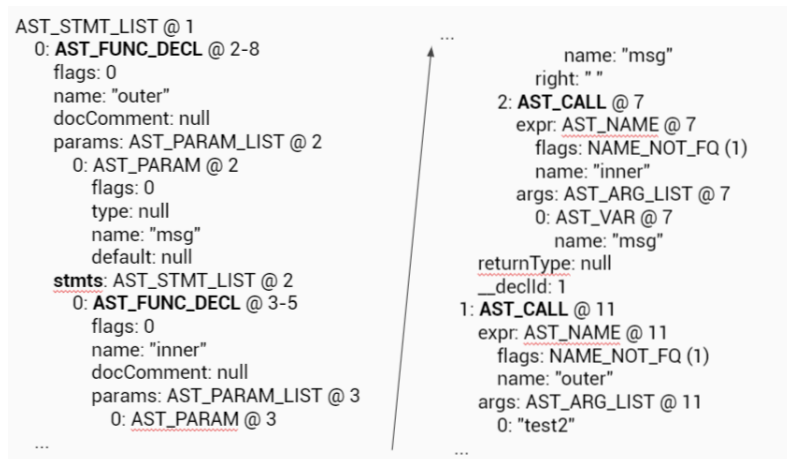


Figure 4.1: Excerpt from an AST file showing function declarations and function calls

Definition 4.3.1 (subtree). A subtree of a tree T is a tree consisting of a node in T and all of its descendants in T (See Figure 4.3).

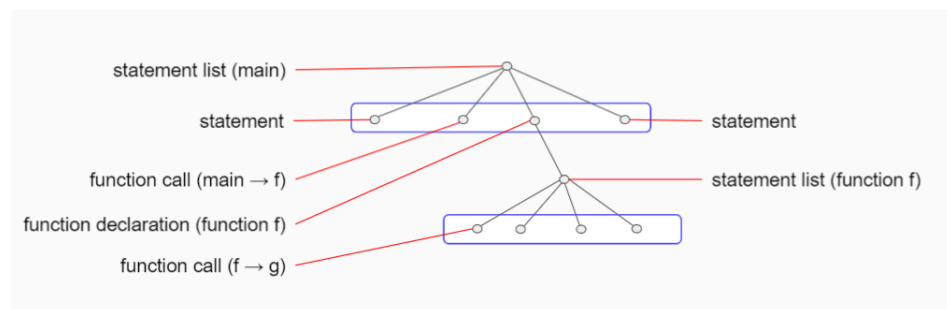


Figure 4.2: Example of function declarations and function calls in an AST

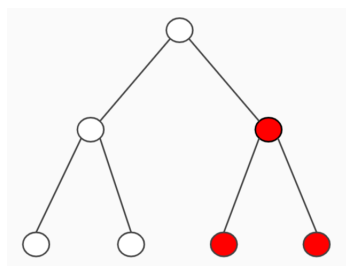


Figure 4.3: The red nodes form a subtree

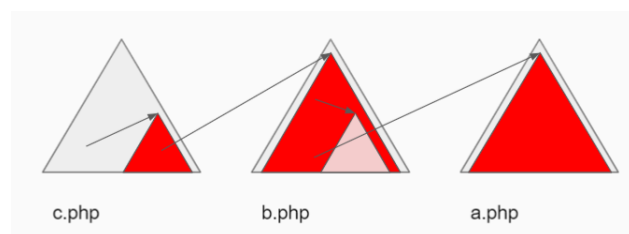


Figure 4.4: ASTs with multiple function calls

When all the source code files of a project have been read, a forest of trees exists. The entry point is chosen in some AST and at another point an exit point determined. That exit point can be in an other function or even in a function in another tree.

The statements of 'main' or of a function are contained in a statement list. In the AST, these are the children of an AST_STMTLIST node. These statements are executed sequentially. The program deviates from this sequence in the case of control structures or function calls.

Each node in the AST has an AST nodetype that indicates which type of statement it is. Examples are:

- AST_ASSIGN
- AST_CALL
- AST_IF
- AST_SWITCH
- AST_WHILE
- AST_FOR

Each statement is defined by the subtree of that node (See Figure 4.3).

A complete list of node types can be found in Appendix C. Some node types may only appear as part of a subtree. For example the node AST_VAR can be a part of a subtree of an AST_ASSIGN statement, to describe to which variable a value is assigned.

Control Flow Statements essentially consist of two types of statements: choices (IF, SWITCH) and loops (WHILE, DO-WHILE, FOR, FOREACH). Branches of choice statements or loops again consist of a statement list. A statement list can in principle be empty (i.e. does not contain any commands), but usually a sequence of statements is executed. With an IF Statement, one branch will be executed (the first branch of which the condition is true). With a SWITCH Statement it is possible to execute combinations of branches. A loop is sometimes not executed, but in theory it can be executed an infinite number of times.

4.4. CREATE A SEARCHABLE DATA STRUCTURE FROM AN AST-FILE

We propose an algorithm that transform the AST into a structure that is suitable for searching for paths in the program. Algorithm B is an algorithm that searches paths directly from the ASTs. We found no source in the literature that described a similar algorithm before.

Algorithm B merges separate CFGs into an interprocedural CFG (ICFG) to provide a structure in which paths can be searched throughout the program. There are earlier descriptions in the literature of procedures which make a large ICFG out of CFG's.

In the general case, the entry point and exit point are located in two different functions. The algorithm must determine how the code path from the entry point runs through the first function and then through a function call to the next function until it reaches the function in which the exit point is located. In the latter function, the code path runs to the exit point. In the case where the entry point and exit point are located in the same function, only paths within that function need to be searched.

Definition 4.4.1 (Simple function path (SFP)). Simple function path from function f to function g is a sequence of function calls starting in s and ending in e . The number func-

tion between s and e might be zero, with 0 or more functions in between. Each function appears at most once in the simple function path.

Definition 4.4.2 (Local Simple Path (LSP)). A local simple path in an AST between two nodes A and B is a simple code path in a single function (without cycles). The path consists of a sequence of nodes in an AST that represent a code path from statement A to statement B. Between A and B are 0 or more nodes of the AST. Each node occurs at most once in the local simple path. In its simplest form, an LSP is a sequence of nodes from a single statement list.

Definition 4.4.3 (Lowest common ancestor). The lowest common ancestor (LCA) of two nodes v and w in a tree or directed acyclic graph (DAG) T is the lowest (i.e. deepest) node that has both v and w as a descendant, where we define each node as a descendant of itself (i.e. if v has a direct connection to w , w is the lowest common ancestor). (See Figure 4.5)

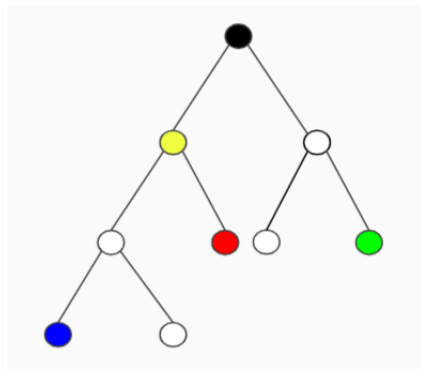


Figure 4.5: The yellow node is the lowest common ancestor (LCA) of the red and blue node.

In an AST the LCA is always a AST_STMTLIST node (See figure 4.6).

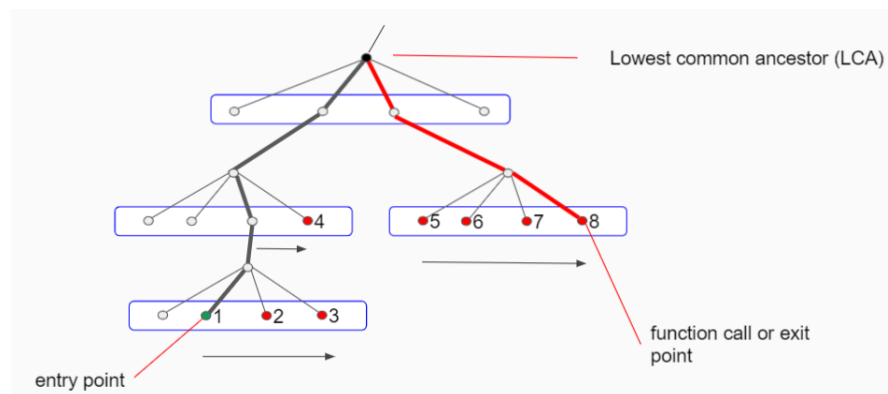


Figure 4.6: In an AST the LCA is a AST_STMTLIST node

4.4.1. ALGORITHM B

This algorithm aims to represent the source code in a structure, in which paths between entry and exit points can be searched. The algorithm immediately processes an AST into an ICFG. If we encounter a function declaration, we remember where it is in the AST. If we

later encounter a function call, we use the node of the declaration to insert the code of the function. The AST's nodes contain all the information about the statements that make up the program. We now describe how to use the AST nodes to build a CFG.

4.4.2. STATEMENTS

There are three types of statements. The first type are the control flow or conditional statements. This category includes IF, SWITCH, WHILE, DO, FOR and FOREACH. We can recognise which statement it is by the AST node type. With these nodes, a condition is evaluated. Depending on the result, the statements in a statement list are executed. A statement list may consist of only one statement. In a CFG, we recognise the control flow statements as nodes from which more than one arrow departs. For each arrow, there is a condition under which the statement list is executed. A condition is an expression that has the value true or false.

Assignments are the second type of statement. Once the code flow arrives at an assignment, it is always executed. An expression is evaluated and the result is put into a variable. The same goes for the third kind of statements: The function calls. With a function call. In PHP, functions may or may not return a function value. If the function returns a function value, the function is usually part of an expression. The expression can be part of a conditional statement (and thus determine the code flow) or it can be part of an assignment; the return value is assigned to a variable. Functions without a return value contain a statement list that can be inserted one by one in the place of the function call.

4.4.3. ASSIGNMENTS

As we have seen, an assignment consists of an expression and a variable value that stores the value of the expression. For a normal concrete execution of a statement, this is the end of the matter. For a symbolic execution we need to look a little further. After executing an assignment, the state of the symbolic store is altered. Suppose the assignment $a=1$ was executed. The variable a then has the concrete value 1, but we can also see this as that the condition $a==1$ is true. In the case of an assignment with a symbolic store, this is no different. Suppose we assign the variable v the symbolic value S , then after execution the assignment the condition $v==S$ is true.

In the usual definition of a CFG [10], the non-conditional statements are summarised in a simple block. This does not match with the way assignments are represented in the ast-file. In the AST each assignment node represents one single expression (See Figure 4.7).

$x=(a-2)*(3+b)$ ○

Figure 4.7: A single assignment in a CFG for a single expression *simple block*

In our algorithm, we do not summarise assignments in a simple block nor as a node per expression, but we include each assignment as one or more separate nodes in the CFG. There are several reasons to do this. Firstly we want each node in the graph to be responsible for generating part of the conditions for the SMT solver. Later on we will see that function calls that are part of an expression are inserted in the graph. These function calls might include conditional statement and loops, which cannot be represented by a simple sequence of nodes.

So in our algorithm an assignment is broken up into separate nodes for each (unary or binary) operator in the calculation. Each intermediate result is stored in a newly introduced result variable that is reused in the next calculation (See Figure 4.8). These result variables are then used in the following conditions to form the overall condition. This is similar to the way that compilers break up the calculation of expressions [1]. The result of $c=(b+7)*(5-a)$ can be broken up as:

```
res1 = b + 7
res2 = 5 - a
res3 = res1 * res2
c = res3
```

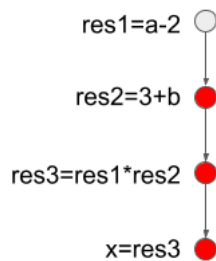


Figure 4.8: In our algorithm assignments in a CFG are a series of nodes for partial calculations

4.4.4. THE WHILE STATEMENT

A WHILE statement is usually included in a CFG in the manner of figure 4.9.

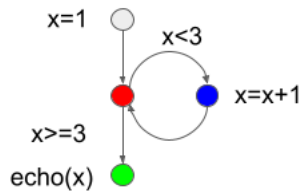


Figure 4.9: The arrows after a WHILE statement are labelled with the conditions.

We will now look at how to include a WHILE statement in a CFG based on its representation in an AST. We use the program in listing 4.4.

```

1 $x=1;
2 while $x<3{
3     $x=$x+1;
4 }
5 echo $x;
```

Listing 4.4: PHP code of an example of a simple WHILE statement

The AST of this program is shown in Listing 4.5.

```

1  AST_STMT_LIST @ 1
2      0: AST_ASSIGN @ 2
3          var: AST_VAR @ 2
4              name: "x"
5          expr: 1
6      1: AST_WHILE @ 3
7          cond: AST_BINARY_OP @ 3
8              flags: BINARY_IS_SMALLER (20)
9              left: AST_VAR @ 3
10                 name: "x"
11             right: 3
12          stmts: AST_STMT_LIST @ 3
13              0: AST_ASSIGN @ 4
14                  var: AST_VAR @ 4
15                      name: "x"
16                  expr: AST_BINARY_OP @ 4
17                      flags: BINARY_ADD (1)
18                      left: AST_VAR @ 4
19                          name: "x"
20                      right: 1
21              2: AST_ECHO @ 6
22                  expr: AST_ENCAPS_LIST @ 6
23                      0: AST_VAR @ 6
24                          name: "x"

```

Listing 4.5: part of the AST code of programm 4.4

We see that the `AST_WHILE` node contains a *cond* key describing the condition that must be met to execute the loop. If the loop is not executed, the negation of this condition is. The conditions are expressions. For the same reasons as with the assignment, we do not label an arrow for this condition, but insert additional nodes for partial calculations of these conditions.

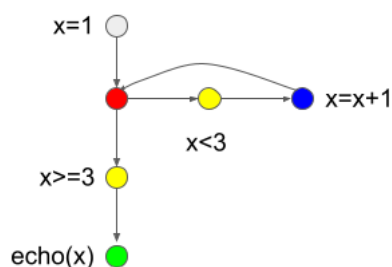


Figure 4.10: The conditions of a WHILE statement are represented by extra nodes before the statement in the loop and after the WHILE

4.4.5. THE IF STATEMENT

As an example of a conditional statement, we will now look at the IF statement. An IF statement always has a branch that is executed if the condition is true. If the condition is false, the IF statement can jump to the first statement after the IF statement or execute an ELSE branch. If ELSEIF branches occur, the associated conditions are included in the same IF statement. We look at this from the program in Listing 4.6.

```
1 if ($t < "12") {  
2   echo "Have a good morning!";  
3 } elseif ($t < "18") {  
4   echo "Have a good day!";  
5 } else {  
6   echo "Have a good night!";  
7 }
```

Listing 4.6: PHP code of an IF statement with multiple branches

For the representation of an IF statement as an CFG, this means the following. For all branches except the ELSE branch, the associated condition is added as a path condition. The arrow pointing to the first node in the branch is responsible for the condition (See Figure 4.11). For the ELSE branch, the negation of all previous conditions is included. If no ELSE branch is present, we must add this branch with an empty statement list.

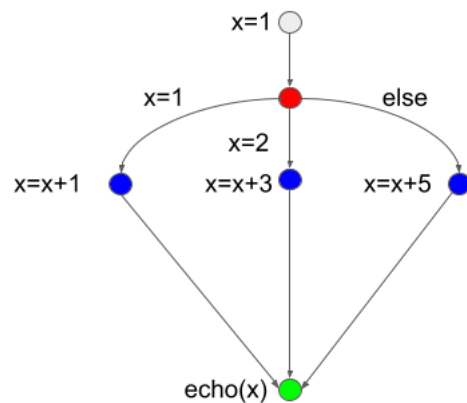


Figure 4.11: Nodes in a CFG resulting from an IF statement

In an AST_IF, all branches are included as an AST_IF_ELEM. The "normal" IF condition and the ELSEIF conditions are described by an expression in the value subtree of the *cond* key. The ELSE branch has a null valued *cond* key 4.7.

```
1      1: AST_IF @ 4
2          0: AST_IF_ELEM @ 4
3              cond: AST_BINARY_OP @ 4
4                  flags: BINARY_IS_SMALLER (20)
5                  left: AST_VAR @ 4
6                      name: "t"
7                      right: "12"
8              stmts: AST_STMT_LIST @ 4
9                  0: AST_ECHO @ 5
10                     expr: "Have a good morning!"
11      1: AST_IF_ELEM @ 6
12          cond: AST_BINARY_OP @ 6
13              flags: BINARY_IS_SMALLER (20)
14              left: AST_VAR @ 6
15                  name: "t"
16              right: "18"
17          stmts: AST_STMT_LIST @ 6
18              0: AST_ECHO @ 7
19                 expr: "Have a good day!"
20      2: AST_IF_ELEM @ 8
21          cond: null
22          stmts: AST_STMT_LIST @ 8
23              0: AST_ECHO @ 9
24                 expr: "Have a good night!"
```

Listing 4.7: Typical AST code for an IF statement

As with the assignment statement, we choose to break up the expression of conditions in our algorithm into separate calculations for the same reasons. Each of the calculations is inserted into the branch as a series of separate nodes before the first statement (See Figure 4.12).

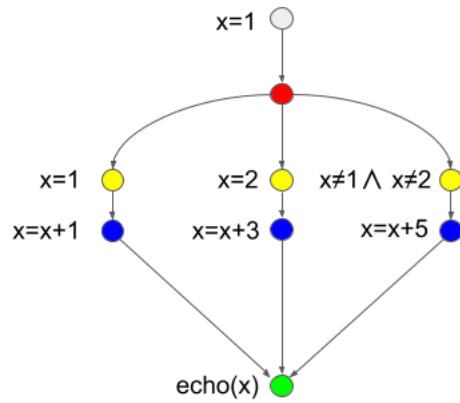


Figure 4.12: Conditions for branches of a conditional statement in a CFG are inserted as extra nodes before the statements.

4.4.6. SWITCH

A Switch statement can be seen as an IF statement with multiple branches. The statement behaves like an IF if a branch is closed with a BREAK (See figure 4.13).

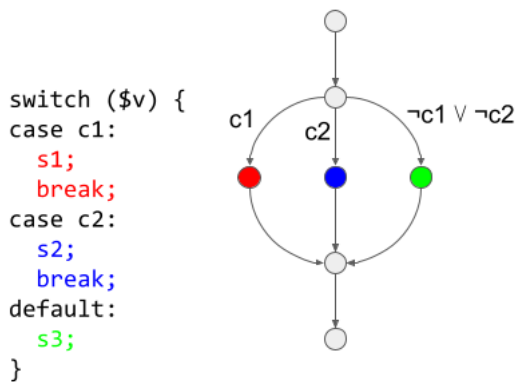


Figure 4.13: CFG of a Switch statement with all branches closed with a BREAK

If the BREAK is omitted, the code flow continues on the next branch (See figure 4.14).

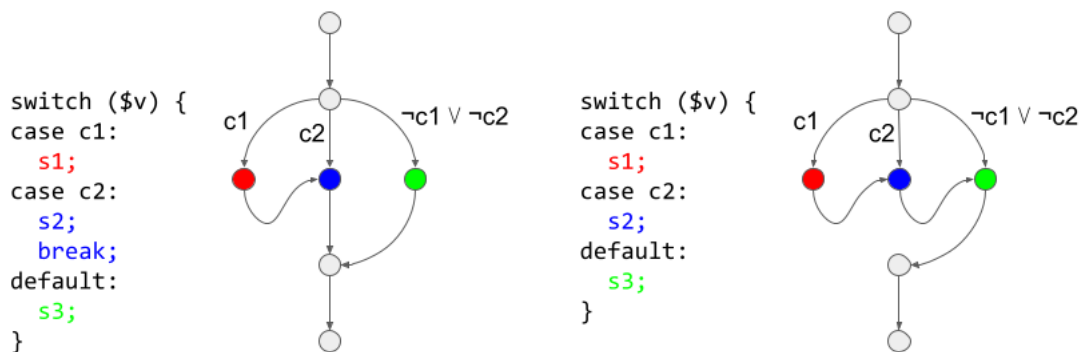


Figure 4.14: CFG of Switch statements with 1 or 2 omitted BREAK statements.

4.4.7. FUNCTIONS

An AST may contain function declarations and function calls. These can be identified by AST_FUNC and AST_CALL node types. In children of the AST_FUNC node, the function name, a parameter list and a statement list are given.

In listing 4.8 we see three examples of function declarations. A normal function (outer1) with an inner function (inner1) inside. In outer2 an anonymous function (inner2) is declared.

```
1  <?php
2  function outer1( \ $msg ) {
3      function inner1( $msg ) {
4          echo( 'inner 1: ' . \ $msg . '\n' );
5      }
6      echo( 'outer 1: ' . $msg . '\n' );
7      inner1( \ $msg );
8      outer2 ();
9  }
10
11 function outer2 () {
12     $inner2=function () {
13         echo( "inner2 \n" );
14     };
15     echo( 'outer 2 \n' );
16     $inner2 ();
17 }
18
19 outer1( 'test outer 1' );
20 outer2 ();
21 ?>
```

Listing 4.8: Example of PHP with three kinds of function declarations

The functions `outer1` and `inner1` can be recognised in the AST by the `AST_FUNC` button. The children `name`, `params` and `stmts` describe the name, parameters and statements respectively.

```

1  <?php
2  AST_STMT_LIST @ 1
3      0: AST_FUNC_DECL @ 2-9
4          flags: 0
5          name: "outer1"
6          docComment: null
7          params: AST_PARAM_LIST @ 2
8              0: AST_PARAM @ 2
9                  flags: 0
10                 type: null
11                 name: "msg"
12                 default: null
13             stmts: AST_STMT_LIST @ 2
14                 0: AST_FUNC_DECL @ 3-5

```

Listing 4.9: Example of PHP with three kinds of function declarations

The anonymous function is described in the assignment. The function has no name, as it is an anonymous function. The value of the key `name` is `"closure"`. This gives the impression that the function is a closure, but this is not the case, as these anonymous functions can only be used within the scope in which they were created.

```

1      0: AST_ASSIGN @ 12
2          var: AST_VAR @ 12
3              name: "inner2"
4          expr: AST_CLOSURE @ 12-14
5              flags: 0
6              name: "{closure}"
7              docComment: null
8              params: AST_PARAM_LIST @ 12
9              uses: null
10             stmts: AST_STMT_LIST @ 12
11                 0: AST_ECHO @ 13
12                     expr: "inner2"
13                     "
14             returnType: null
15             __declId: 2

```

Listing 4.10: Example of PHP with three kinds of function declarations

For the AST_CALL node, the AST describes which function is called and which arguments are substituted for the parameters.

```

1      2: AST_CALL @ 19
2          expr: AST_NAME @ 19
3              flags: NAME_NOT_FQ (1)
4              name: "outer1"
5          args: AST_ARG_LIST @ 19
6              0: "test outer 1"

```

Listing 4.11: Example of PHP with three kinds of function declarations

The closure is in fact a function variable and is therefore called in the same way as if a normal function had been assigned to a variable.

```

1      2: AST_CALL @ 16
2          expr: AST_VAR @ 16
3              name: "inner2"
4          args: AST_ARG_LIST @ 16

```

Listing 4.12: Example of PHP with three kinds of function declarations

```

1  <?php
2  function f($a,$b ) {
3      $z=$a*$b;
4      return $z;
5  }
6
7  $x=1;
8  $y=2;
9  $z=f($x,$y);
10 echo($z);
11 ?>

```

Listing 4.13: Example of a PHP program with a function call

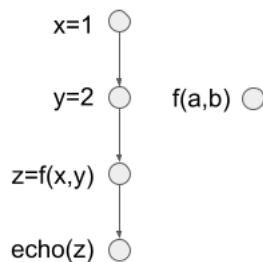


Figure 4.15: A node with a function call, calling the function f(a,b) with the arguments x and y.

When inserting the function into the CFG, the arguments for the parameters must be entered. These are followed by the nodes of the statements of the function.

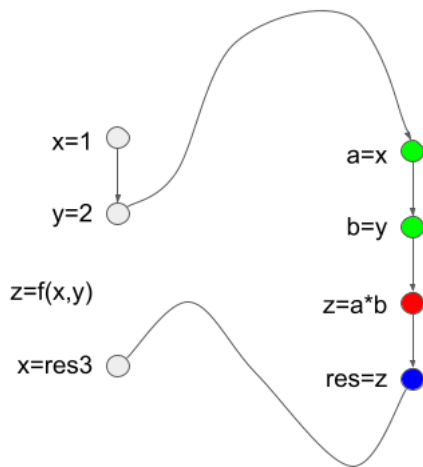


Figure 4.16: Example of a CFG where the function call is replaced by a parameter substitution and the CFG of the function.

4.4.8. FUNCTIONS CALLING FUNCTIONS

If a function calls another function, this can give rise to a - in principle - infinitely long series of function calls. An example is a function that calls itself: a recursive function. Fibonacci numbers are a well-known example of numbers which can be calculated with a recursive function (See Listing 4.14).

```

1  <?php
2  // Recursive function for Fibonacci series.
3  function Fibonacci($number){
4
5      // the first two numbers in the series
6      // are 0 and 1
7      if ($number <= 1)
8          return $number;
9
10     // Recursively call the function Fibonacci
11     // for the previous to numbers in the series
12     else
13         return (Fibonacci($number-1) + Fibonacci($number-2));
14 }
15
16 $number = 10;
17 for ($counter = 0; $counter < $number; $counter++){
18     echo Fibonacci($counter), ' ';
19 }
20 ?>

```

Listing 4.14: Example of a recursive function in PHP

If it is possible in a program for functions to keep calling each other in an infinite sequence, this causes the algorithm composing the CFG to never end. In the case of the Fibonacci function listing 4.14, the algorithm keeps inserting the function forever.

This can be avoided by setting a limit on the "depth" at which functions may call another function. We call this depth the recursion depth, although it also applies in the case where a function other than itself is called. If we set the recursion depth to 10, the function will not be inserted after 11 times.

We need to determine what should happen at the point where the algorithm stops inserting CFGs of functions. We can just skip the function to be inserted. As a result, it is not clear whether a path passes this point. If we add a node at the point where the CFG of the function should have been inserted, this node, when compiling the SMTLIB code, can add an assertion showing that the point was in the path: (assert (= maxrecursiondepth_001 True)). Because these types of points can lie in a path more than once, a counter must be added to prevent the same variable from being used more than once.

4.4.9. FUNCTIONS VARIABLES

We have seen before that in PHP we can assign a function name to a variable. This variable can then be used to call that (variable) function. We call such a variable a function variable.

FUNCTION VARIABLES USING A CONCRETE FUNCTION NAME

If a function variable contains a concrete function name, the CFG of this function can be inserted in the same way as a normal function call. If the function variable depends on a variable, the function can only be inserted correctly if the value of this variable is known. However, the value of the variable may depend on the actual path followed.

FUNCTION VARIABLES USING A SYMBOLIC FUNCTION NAME

The value assigned to a function variable may also be a symbolic value. This situation is complex. Which functions qualify for this symbolic value depends on a number of factors. In principle, all functions are eligible. But a limiting factor could be the number of parameters: the number of arguments to the function call must match the number of parameters. We then limit the set of eligible functions to those with a matching number of parameters. We can then examine several variants of a path that differ in the function substituted for the symbolic function name. However, it is possible in PHP to write functions whose number of parameters is variable. In those functions, a developer can write PHP functions¹ that can request the number of parameters, so that the function can execute a command depending on that number. We have not gone down this complicated path.

¹<https://www.php.net/class.reflectionfunction>

4.4.10. ARRAY'S

An array in PHP is an ordered map: a list of key, value pairs. This data structure can be used to form data structures known in other programming languages as indexed arrays, associative arrays, lists or maps.

Integers or strings can be used as keys. In an array, keys of integer type and string type can be used interchangeably. The values can be of any PHP type.

In its simplest form, we assign a number of integer or string values to an array. PHP then chooses the keys itself. If there are no elements in the array yet, PHP chooses zero as the first index. If there are already elements in the array, PHP chooses the highest index plus one. This means that some indexes may remain unused.

The `array()` keyword can be used to assign an empty array to a variable.

```
$arr1=array();
```

In the AST we recognise this with the `AST_ARRAY` node as in the listing 4.15.

```
1      3: AST_ASSIGN @ 9
2          var: AST_VAR @ 9
3              name: "arr2"
4      expr: AST_ARRAY @ 9
5          flags: ARRAY_SYNTAX_LONG (2)
6      0: AST_ARRAY_ELEM @ 9
7          flags: 0
8          value: 1
9          key: null
10     1: AST_ARRAY_ELEM @ 9
```

Listing 4.15: Example of a recursive function in PHP

By placing values between the brackets of the keyword `array()`, we can immediately fill the array. This can be done with integer values, string values or any other PHP type. The first element from the list of values is assigned index 0, the second index 1 and so on.

```
$arr2=array( 1,2,3 );
$arr3=array( "a","b","c" );
```

The `AST_DIM` key of the assignment indicates that an index in an array follows. The name of the array is found at the `expr` key, the index at the `dim` key (see listing 4.16).

```
1      4: AST_ASSIGN @ 12
2          var: AST_VAR @ 12
3              name: "arr2"
4      expr: AST_ARRAY @ 12
5          flags: ARRAY_SYNTAX_LONG (2)
6      0: AST_ARRAY_ELEM @ 12
7          flags: 0
8          value: 1
9          key: null
```

Listing 4.16: Example of a recursive function in PHP

We can change the value of an existing index with an assignment. If the index of the assignment does not exist yet, the first free integer value is chosen which is greater than the highest index so far. If no index is written between the brackets, PHP will choose the index according to the same procedure.

```
$arr1[2]=3;      // 2 will be the first index in $arr
$arr2[3]=2;      // adding an element
$arr3[2]="b";    // replacing an element
```

In the AST we see that the array element to which a value is assigned is in the AST_DIM node. The expr key contains the name of the array and the dim key contains the index (see listing 4.17).

```
1      4: AST_ASSIGN @ 10
2          var: AST_DIM @ 10
3              flags: 0
4              expr: AST_VAR @ 10
5                  name: " arr2 "
6                  dim: 3
7              expr: 2
```

Listing 4.17: Example of a recursive function in PHP

Conversely, we can use an element of an array in an expression, for example, to assign the value to another variable.

```
$var = $arr1[1];
```

In the AST, the same method is used for indexing, only now it is not in the var key of the assignment, but in the expr key (see listing 4.18).

```
1      3: AST_ASSIGN @ 8
2          var: AST_VAR @ 8
3              name: " var "
4          expr: AST_DIM @ 8
5              flags: 0
6              expr: AST_VAR @ 8
7                  name: " arr1 "
8              dim: 1
```

Listing 4.18: Example of a recursive function in PHP

An associative array is filled by specifying the key, value pair. The key can be an integer or a string. The value can be of any PHP type. We will only look at an example with integers and strings.

```
$things = array("a"=>1, 2=>"23", "Joe"=>"34", 4=>5);
```

In the AST, we now see that for each AST_ARRAY_ELEM the value and the key are given a value (see listing 4.19).

```
1      17: AST_ASSIGN @ 39
2          var: AST_VAR @ 39
3              name: "things"
4          expr: AST_ARRAY @ 39
5              flags: ARRAY_SYNTAX_LONG (2)
6              0: AST_ARRAY_ELEM @ 39
7                  flags: 0
8                  value: 1
9                  key: "a"
10             1: AST_ARRAY_ELEM @ 39
11                 flags: 0
12                 value: "23"
13                 key: 2
```

Listing 4.19: Example of a recursive function in PHP

4.4.11. GRAPHS RESULTING FROM ALGORITHM B

Figure 4.12 shows an example of a CFG constructed with Algorithm B.

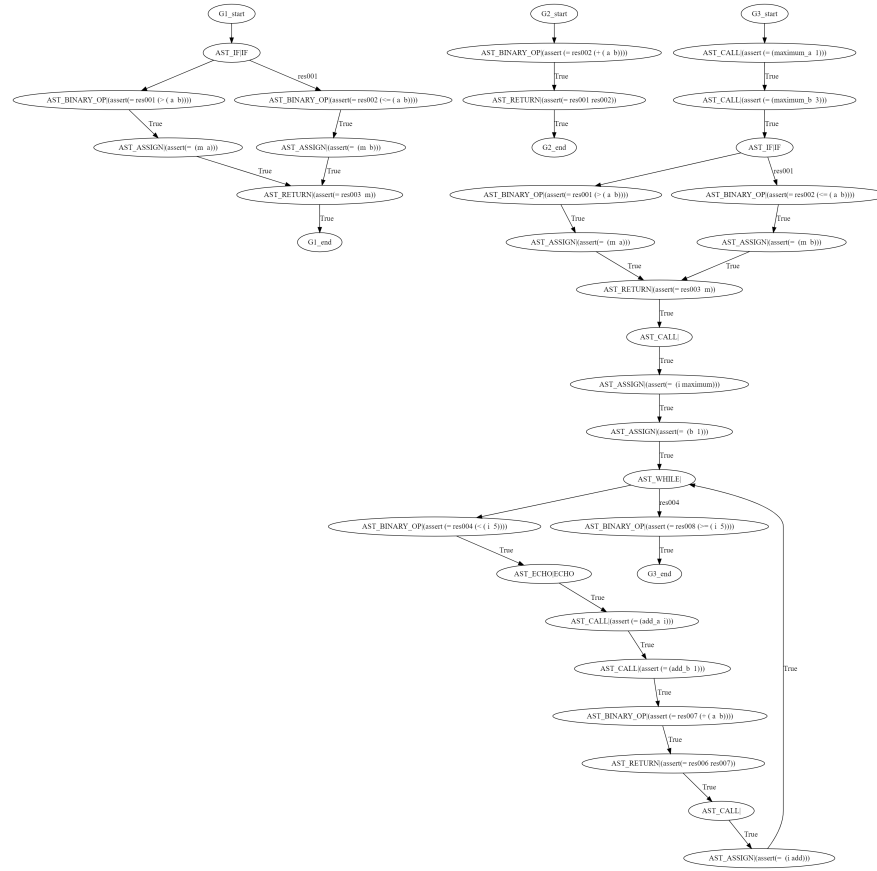


Figure 4.17: Example of a CFG constructed with algorithm B

4.5. CONSTRUCTION OF PATHS

Now that we have a CFG representing the code, we can search for paths between entry point E and exit point X in the CFG. In a CFG with a conditional statement, there can be more than one path between an entry point and an exit point. In the case of an IF statement, there are at least two paths. In a WHILE loop there can be infinitely many paths between E and X (See listing 4.20).

```
1  <?php
2  $x = 0;
3  while($x <> 1) {
4      $x = random_int(1,1000000);
5      echo($x. " ");
6  }
7  echo($x);
8  ?>
```

Listing 4.20: Example of a PHP program with a while loop that possibly takes a while

Both the Breadth First Search (BFS) and Depth First Search (DFS) algorithms can, with appropriate modification, generate useful paths in our situation. BFS is useful if we are looking for short paths. DFS is more useful if we expect the paths to be long.

Our algorithm assumes an entry point E and a set of exit points X. The steps of the algorithm construct a search tree of all paths leading from the root to the exit points. Whether a point is added from a given node is determined by a feasibility criterion. If the algorithm can decide that the stop criterion is no longer feasible, the search tree is no longer extended at that point. When an exit point is added to the search tree, a stop criterion decides if the path from the root to this point is added to the set of solutions. Thus, we do not assume that a path necessarily ends when it reaches an exit point. If the exit point lies in a loop, it may be that we are looking for a path which first passes the exit point a number of times before the stop criterion is fulfilled (e.g. a vulnerability is triggered).

The stopping criterion takes into account two characteristics of the path: the number of times a node appears in the path and the length of the path. The length of the path π must be an element of a set of admissible lengths $L(\pi)$ for the paths π . For example, we can require that $L(\pi) = \{x | x < 20\}$.

The number of occurrences of a node v_i in a path π is written as $f_\pi(v_i)$. In the starting situation, an empty tree, all frequencies are zero. If the node v_i is added to the search tree, we increase the frequency of $f_\pi(v_i)$ by 1. The target consists of a set with per node a set of target frequencies $tf_\pi(v_i)$. If there are no special requirements for a node (the number of times a node occurs in the path is free), we take $tf_\pi(v_i) = \mathbf{N}$. If we require that a node must occur 1 or 2 times in the path we require $tf_\pi(v_i) = \{1, 2\}$.

If we want only simple paths (cycle free paths) as a solution we take the target frequency for all nodes to be $tf_\pi(v_i) = \{1\}$. The path may then enter the loop. This can be avoided by giving the first node in the loop the target frequency $tf_\pi(v_i) = \{0\}$. The stop criterion is fulfilled if for all nodes in the path π it holds that $f_\pi(v_i) \in tf_\pi(v_i)$

As mentioned above, all frequencies $f_\pi(v_i)$ are zero in the initial state. When adding a node, the frequency can only increase. The goal is feasible, as long as the frequency of all nodes in the path is less than or equal to the maximum frequency of the set target frequency

of that node is. For all nodes in the path π , $f_\pi(v_i) < \max(t f_\pi(v_i))$. Therefore, $\max(t f_\pi(v_i))$ is the highest value in the set $t f_\pi(v_i)$.

The paths that satisfy the conditions form a solution set S . The set S is empty in the initial state. If at an exit point the goal is reached, the path leading from the entry point to the exit point is added to the set S .

It is important to realise that the algorithm generates cycle-free paths.

The algorithm proceeds according to the steps in algorithm 1.

Data: Given: Graph G , Entry point E , Exit point X , Target T

```

 $S \leftarrow \emptyset$ ;
Tree.create( $E$ );
Tree.setTarget( $T$ );
 $Q \leftarrow E$ ;
while  $Q \neq \emptyset$  do
  forall  $q_i \in Q$  do
     $Q.remove(q_i)$ ;
    if  $T.TargetFeasibleFrom(q_i)$  then
      forall  $b_j \in CFG.neighbours(q_i)$  do
        Tree.addChild( $q_i, b_j$ );
        if  $q_i \in X \wedge Tree.TargetReachedIn(b_j)$  then
           $S.add(\pi)$ ;
        end
         $Q'.add(b_j)$ ;
      end
    end
  end
   $Q \leftarrow Q'$ 
end

```

Algorithm 1: The search algorithm

As an example, we consider the CFG in 4.18 with six nodes. The graph consists of the six nodes $v_1, v_2, v_3, v_4, v_5, v_6$. The entry point is v_1 . The set of exit points is v_6 . The collection of target frequencies is: $v_1 = \mathbf{N}, v_2 = \mathbf{N}, v_3 = 1, v_4 = \mathbf{N}, v_5 = \mathbf{N}, v_6 = 1$. For path length, the target is \mathbf{N} (any value is allowed).

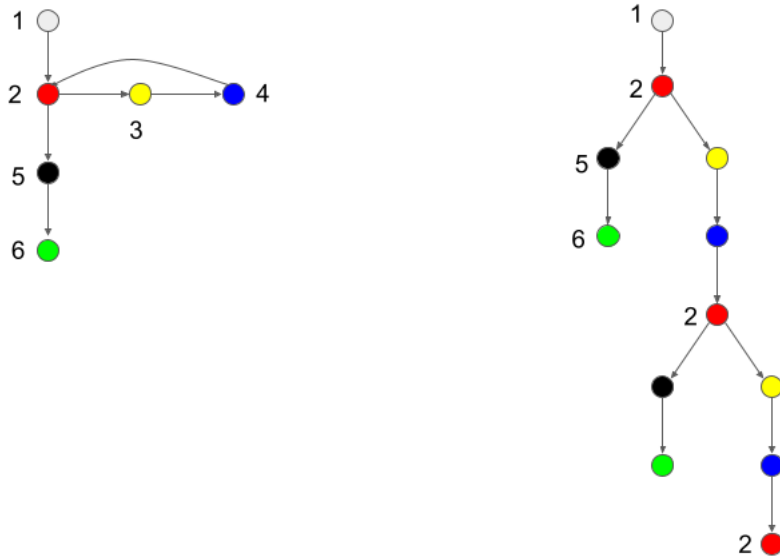


Figure 4.18: On the left: a small CFG to illustrate the search algorithm. On the right: the resulting search tree.

Preparation

Step 1: Set the frequencies for all nodes to 0.

Step 2: Put the entry point v_1 in the queue. Adjust the frequency of this node: $f(v_1) = 1$.

Round 1

Step 1: The queue is not empty, it contains v_1 . Make newQueue empty.

Step 2: Retrieve v_1 from the queue.

Step 3: The target is feasible in v_1 , because for all nodes the frequency is less than or equal to the maximum target frequency of that node. Therefore, determine which nodes are the neighbours of v_1 : only v_2 is a neighbour.

Step 4: Expand the search tree to include the branch $v_1 \rightarrow v_2$.

Step 5: Since v_2 is not an exit point, we have not reached an endpoint of a path.

Step 6: Enter v_2 to newQueue

Step 7: Because the queue is empty, copy newQueue to Queue

round 2

Step 1: The queue is not empty, it contains v_2 . Make newQueue empty.

Step 2: Get v_2 from the queue

Step 3: The target is feasible in v_2 , because for all nodes the frequency is less than or equal to the maximum target frequency of that node. Therefore, determine which nodes are the neighbours of v_2 : v_3 and v_5 are neighbours.

Step 4: First look at v_3 . Expand the search tree to include the branch $v_2 \rightarrow v_3$.

Step 5: Since v_3 is not an exit point, we have not reached an endpoint of a path.

Step 6: Enter v_3 to newQueue

Step 7: Now look at v_5 . Expand the search tree to include the branch $v_2 \rightarrow v_5$.

Step 8: Since v_5 is not an exit point, we have not reached a possible endpoint of a path.

Step 9: Enter v_5 to newQueue

Step 10: Since the queue is empty, we copy newQueue to Queue

round 3

Step 1: The queue is not empty, it contains v_3 and v_5 . Make newQueue empty.

Step 2: Take v_3 out of the queue

Step 3: The target is feasible in v_3 , because for all nodes the frequency is less than or equal to the maximum target frequency of that node. Therefore, determine which nodes are the neighbours of v_3 : v_4 is a neighbour.

Step 4: Consider v_4 . Expand the search tree to include the branch v_4 .

Step 5: Since v_4 is not an exit point, we have not reached a possible endpoint of a path.

Step 6: Feed v_4 to newQueue.

Step 7: Retrieve v_5 from the queue

Step 8: The goal is feasible in v_5 , because for all nodes the frequency is less than or equal to the maximum goal frequency of that node. Therefore, we determine which nodes are the neighbours of v_5 : v_6 is a neighbour.

Step 9: Consider v_6 . Expand the search tree to include the branch v_5 .

Step 10: Since v_6 is an exit point, we have reached a possible endpoint of a path. Because for all nodes the frequencies are elements of the target frequencies of that node (we have reached a goal), we add the path leading from v_1 to v_6 in the search tree to the solutions: $v_1 \rightarrow v_2 \rightarrow v_5 \rightarrow v_6$

Step 11: Enter v_6 to newQueue

Step 12: since the queue is empty, we copy newQueue to Queue

round 4

Step 1: The queue is not empty, it contains v_4 and v_6 . Make newQueue empty.

Step 2: We take v_4 out of the queue

Step 3: The target is feasible in v_4 , because for all nodes the frequency is less than or equal to the maximum target frequency of that node. Therefore, determine which nodes are the neighbours of v_4 : v_2 is a neighbour.

Step 4: Consider v_2 . Expand the search tree to include the branching v_4 .

Step 5: Since v_2 is not an exit point, we have not reached a possible endpoint of a path.

Step 6: Input v_2 to newQueue

Step 7: Retrieve v_6 from the queue

Step 8: The goal is feasible in v_6 , because for all nodes the frequency is less than or equal to the maximum goal frequency of that node. Therefore, determine which nodes are the neighbours of v_6 : There are no neighbours. Therefore, we are finished processing v_6

Step 9: Since the queue is empty, we copy newQueue to Queue.

The only path resulting from the algorithm is: $v_1 \rightarrow v_2 \rightarrow v_5 \rightarrow v_6$

5

EVALUATING PHP CODE BY SYMBOLIC EXECUTION

5.1. RQ2: UP TO WHAT EXTENT CAN PHP-CODE BE EVALUATED USING SYMBOLIC EXECUTION?

As a result of RQ1, we found paths. The nodes in these paths describe state changes that we can express as conditions for an SMT solver. We use the intermediate language SMTLIB for this purpose, because it allows us to make the expressions suitable for other SMT solvers besides the Z3 solver we use in this study.

A PHP program consists of a number of statement types: assignments, function calls, loops, conditional statements.

We adopted the assignments when processing into a CFG as a series of nodes describing the sub-operation. We added the conditions of loop and conditional statement to the path. User defined functions we inserted at the place of function call. Library functions are to be processed as best as possible into conditions that describe the state change that the function call brings about.

We first look at how to write variables, conditions and assignments in SMTLIB. We have to take into account that a variable can be assigned a value more than once. Next, we will look at how library functions can be evaluated. Finally, we will look at the problem of determining the type of a variable.

5.2. VARIABLES IN SMTLIB

In contrast to PHP, SMTLIB variables must be declared. They are introduced in an assignment or as parameters of a function. A variable has a name (identifier) and a type.

If a variable is not introduced first, an error message will be returned when the program is executed. The following code declares the variable amount of type Int.

```
(declare-const amount Int)
```

Apart from the type Int, other types are possible in SMTLIB: Real, String, Bit vector, Bool and Array. We must ask ourselves to what extent the types of SMTLIB correspond to the types in PHP. In addition, the type chosen determines which operators are available.

SMTLIB type	comparison
String	The string type of PHP ¹ and SMTLIB ² corresponds. In both languages it is possible to work with unicode strings.
Bitvector ³	In PHP, integers are stored in integer variables. The documentation indicates that integer elements from the infinite collection \mathbb{Z} . In fact the number of bits available for an integer depends on the system and is limited to 32 or 64 bits ⁴ . In SMTLIB, bit vectors can be used whose length can be freely selected.
Int	In terms of characters, SMTLIB stumbles on two legs. For addition, only an unsigned operator is available (which is not really a problem because the result of signed and unsigned intersections is bitwise the same).
Real	can be used to represent the PHP type integer.
Bool	Corresponds tot the PHP type float
	can be used to represent the PHP boolean type. The result of a condition is a Boolean value (sometimes concretely True or False)

Table 5.1: Comparison of PHP and SMTLIB Types

It seems obvious to link PHP types and SMTLIB types in the way shown in table 5.2. However, the precision of the float/real type is not necessarily the same and therefore the results of concrete and symbolic may differ. The same applies to integers. The PHP type Integer technically corresponds best to a bit vector of 32 or 64 bits, but it may be necessary to use an operator that only occurs with the type Int. For example, SMT has a cast function for integers, but not for bit vectors.

PHP type	Sort (SMTLIB Type)
String	String
Integer	Int or Bitvector
Float	Real
Boolean	Bool
Resource	A resource variables hold special handles to opened files, database connections, image canvas areas and the like. In PHP, converting to a resource makes no sense. ⁵ . In SMTLIB we have no equivalent type.
Array	In addition to arrays of mixed types, SMTLIB has the ability to perform store and select operations with arrays
Class	Mixed form of variables and functions

Table 5.2: Map PHP types to SMTLIB types

The name of a variable is used in the main program, but also in the functions. To distinguish between these use of variables we use the term context. The context is the function name in which the variable is introduced.

The main program does not have a function name, but we use main, as is common in other programming languages such as C and Python. The assignment $\$a=2$; introduces the variable $\$a$ into context main.

```
(declare-const a Int)
```

After executing the assignment, the condition is that the value of a must be equal to two. So we can say that an assignment adds a condition. This is written in SMTLIB as:

```
(declare-const a Int)
(assert (= a 2))
```

A variable can be assigned a value more than once.

```
$a=2;
$a=3;
```

Translating blindly to the condition $a = 2 \wedge a = 3$ we see that it has no solution. If $\$$ has the value 2, the condition $\$=3$ is false, and if $\$$ has the value 3, the condition $\$=2$ is false. Thus, there is no valuation that makes both condition true.

With a counter we keep track of how often a value is assigned to the variable in the path. This is necessary to be able to distinguish between the different assignments that are made to a variable.

The condition $a_1 = 2 \wedge a_2 = 3$ has the solution $a_1 = 2, a_2 = 3$.

This method is sufficient because in the paths we construct with algorithm 1 there are no loops [1].

We use this counter to give the variables in SMTLIB a unique value. Variables can have a symbolic or a concrete value. In this case, the variables are given a concrete integer value. Thus, the variables also become of integer type.

```
(declare-const a_001 Int)
(declare-const a_002 Int)
(assert (= a_001 2))
(assert (= a_002 3))
```

In the code of listing 2.13, we can see that the variables \$x and \$y are assigned a value more than once.

In figure 2.11 we have expressed the path conditions as symbolic variables. The states of program 2.13 when using SSA for numbering variables can be seen in the symbolic execution tree in figure 5.1.

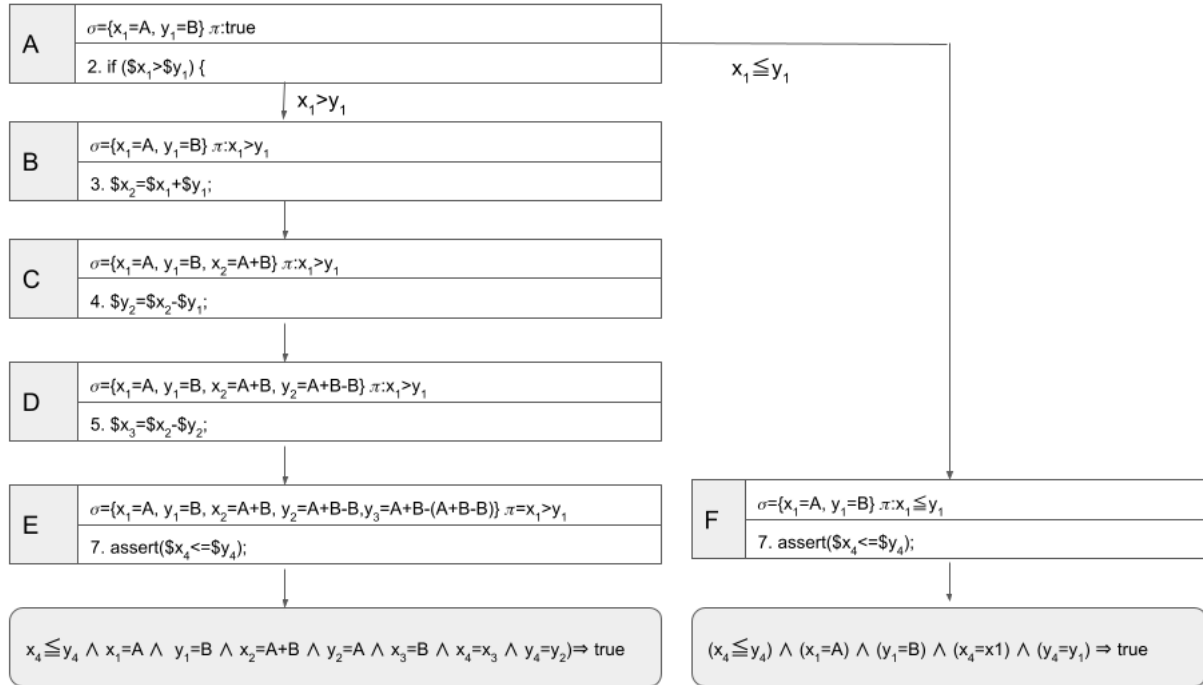


Figure 5.1: Symbolic execution tree with SSA

5.3. CONDITIONS AND ASSIGNMENTS IN SMTLIB

A condition in SMTLIB is written in Polish notation[24]. The condition

$$a = (2 + b) * (1 - a)$$

is written as

$$= a * + 2 b - 1 a .$$

In SMTLIB we write this as

```
(assert (= a_002 (* (+ 2 b_001) (- 1 a_001))))).
```

The counter is added to distinguish between successive assignments. As we have seen before, we can consider an assignment $v=S$ as a condition that is fulfilled after executing the assignment. Therefore we can write an assignment in SMTLIB as $(\text{assert } (= v S))$. For more complicated expressions without a function call, we could also write an assertion.

$$\$c=(5-\$a)*(\$b+7);$$

then becomes

```
(assert (= ( c (* (- 5 a) (+ b 7)))))
```

Which we can break up into:

```
= (res001 (+ (b 7)))
= (res002 (- (5 a)))
= (res003 (* (res001 res002)))
= (c res003)
```

These, in turn, can be translated into an SMTLIB condition. Each condition belongs to a node in the CFG.

```
(assert (= (res001 (+ (b 7)))))
(assert (= (res002 (- (5 a)))))
(assert (= (res003 (* (res001 res002)))))
(assert (= (c res003)))
```

conditions of a conditional statement (IF, SWITCH, WHILE, DO, FOR) are translated to a set of nodes in a similar way. The PHP expression

$$\$a * \$a + \$b * \$b == \$c * \$c;$$

then becomes

```
(assert (= true (= (* c c) (+ (* a a) (* b b)) )))
```

Or as separate statements:

```
(assert (= res001 (* c c)))
(assert (= res002 (* b b)))
(assert (= res003 (* a a)))
(assert (= res004 (+ res002 res003)))
(assert (= res005 (= res004 res001)))
(assert (= true res005))
```

The last two statements can be summarised as

```
(assert (= true (= res004 res001)))
```

5.4. ARRAY'S

In the next section, we look at two ways to execute arrays symbolically. The first solution works with separate integer and string variables. The second method uses the Array Theory of Z3. In the tool we use the first method, but in some examples we show the advantages of using Z3 arrays.

5.4.1. SINGLE VARIABLES

If an array contains a small number of elements that don't change (retaining the symbolic value is an example), we can declare the elements as separate variables. An example of this is the superglobals `$_POST` and `$_GET`. These variables are arrays. These arrays often contain a small number of elements, for example, `$_POST['username']`, which serve as input. In symbolic execution, this input is given a symbolic value. We can, instead of using real arrays, implement a simplification by declaring a variable of type string.

```
(declare-const post_username String)
```

For other small arrays we can use a similar simplification.

```
$arr = array( "tic","tac","toe");
```

For a small array with the concrete elements "tic", "tac", "toe" we can declare three separate string variables, which we then assign the appropriate values.

```
(declare-const arr_tic String)
(declare-const arr_tac String)
(declare-const arr_toe String)
(assert (= arr_tic "tic") )
(assert (= arr_tac "tac") )
(assert (= arr_toe "toe") )
```

When using an array element or assigning a new value to an array element, we can use the names of the array elements in the conditions that apply after the assignment.

The PHP commands

```
$x = $arr["tic"];
$arr["tac"] = "cheese";
```

can thus be translated into:

```
; additional declaration for the variable x
(declare-const x String)

(assert (= arr_tic "tac") )
(assert (= arr_tac "cheese") )
```

Assigning a symbolic variable is the same as for normal variables. If the index is a variable, we cannot determine in advance which element we mean.

```
$value = $arr[ $key ];
```

We can then use SMTLIB’s `ite` (if-then-else) command to determine which variable contains the array element we are looking for (in the example we took a concrete value for the key, but if the key is a symbolic value the method works the same).

[illegible]

The secret is in the generation of the last line. We can also reverse this. If the value of the array element is given, we can ask for the corresponding key. We now create a search function using the `ite` functions that determines the value of the corresponding key.

```
(declare-const arr_tic String)
(declare-const arr_tac String)
(declare-const arr_toe String)

(declare-const key Int)
(declare-const value String)

(assert (= value "tac") )

(assert (= arr_tic "tic") )
(assert (= arr_tac "tac") )
(assert (= arr_toe "toe") )

(assert (= key (ite (= value "tic")
                    0 (ite (= value "tac")
                            1 (ite (= value "toe")
                                    2 "-1"))))))
))))))
```


The method mentioned above works for array where key and value are of type integer or string (with matching declarations). As mentioned, the method works well for arrays that are not modified. The key and value pairs are constant and therefore the if-then-else statements can be used to compose a query that fits the situation.

With some effort, we could also change the value of an element. The penalty we pay for this approach is that we must create a new ssa version for all array elements, because after the ite statements the code does not know for which element the conditions have changed. If the all conditions change, it is clear from that point on that a new version has been created for all elements of the entire array.

```
(declare-const arr_0_001 Int)
(declare-const arr_1_001 Int)
(declare-const arr_2_001 Int)
(declare-const arr_0_002 Int)
(declare-const arr_1_002 Int)
(declare-const arr_2_002 Int)

(declare-const key Int)
(declare-const value Int)

; fill array
(assert (= arr_0_001 2) )
(assert (= arr_1_001 3) )
(assert (= arr_2_001 5) )

; set array[key] to value value
(assert (= key 1) )
(assert (= value 7) )

; set the right element to value 7
(assert (= arr_0_002 (ite (= key 0) value arr_0_001)))
(assert (= arr_1_002 (ite (= key 1) value arr_1_001) ))
(assert (= arr_2_002 (ite (= key 2) value arr_2_001) ))
```

5.4.2. USING SMTLIB'S ARRAY'S

SMTLIB masters the possibility of declaring arrays. For these arrays we can use integer or string types for the key or value.

```
(declare-const arr1 (Array Int Int))
(declare-const arr2 (Array String Int))
(declare-const arr3 (Array Int String))
(declare-const arr4 (Array String String))
```

The function (store arr index value) can be used to store a value on an index in an array. This function returns a new version of the array in which the corresponding index has changed. With (store ((as const(Array Int Int)) 0), all elements of the array are set to zero.

The PHP code

```
$arr = array( 1,2,3 );
```

can be written in SMTLIB as:

```
(declare-const arr (Array Int Int))
(assert (= arr
  (store
    (store
      (store
        ((as const(Array Int Int))
          0)
        0 1)
      1 2)
    2 3) ))
```

From this array we can determine the value of an element with the (select arr index) command.

```
(declare-const key Int)
(declare-const value Int)
(declare-const arr (Array Int Int))

; fill the array
(assert (= arr
  (store
    (store
      (store
        ((as const(Array Int Int))
          0)
        0 1)
      1 2)
    2 3) ))

;request the value for index 1
(assert (= key 1))

; value, now contains the value at index 1
(assert (= value (select arr index)))
```

This corresponds to an assignment in PHP:

```
$arr = array( 1,2,3 );
$key=1;
$value=$arr[$index];
```

In a similar way, we can change the value of an element of the array. However, we must take into account that we need a second array to store the changed array.

```

(declare-const key Int)
(declare-const value Int)
(declare-const arr1 (Array Int Int))
(declare-const arr2 (Array Int Int))

; fill the array
(assert (= arr
  (store
    (store
      (store
        ((as const(Array Int Int))
          0)
        0 1)
      1 2)
    2 3) ))

;we are about to change the value for index 1
(assert (= key 1))
(assert (= value 7))

; value at index 1 is now 7
(assert (= arr2 (store arr1 index value)))

```

In PHP we could write this as:

```

$arr = array( 1,2,3 );
$key=1;
$value=7;
$arr[$key]=$value;

```

5.5. FUNCTIONS

5.5.1. USER DEFINED FUNCTIONS

A path may contain the code of a called function. This function may contain parameters or variables with the same name as in another parts of the path. We must ensure that these parameters and variables have a unique name.

We solve this problem by making the naming of parameters and variables unique at each call by combining a counter for each call and the name of the parameters or local variables into a unique name. With this naming scheme, we can guarantee unique names for variables even with multiple function calls.

`$functionname_functioncounter_variablename_varcounter`

We illustrate this with the program in Listing 5.1. We have shortened the name of the function to make the example easier to read, so that the names of the variables do not become too long.

```
1 function cs(&$x, &$y) {
2     if ($x>$y) {
3         $x=$x+$y;
4         $y=$x-$y;
5         $x=$x-$y;
6     }
7     assert ($x<=$y);
8 }
9
10 conditionalswap (2 , 1);
```

Listing 5.1: Example of a PHP program that swaps the values of the variables \$x and \$y if \$x>\$y.

There are again two possible paths. When calling the function, we need to enter arguments 2 and 1 for the parameters x and y. We use a counter to keep track of how many times the function has been called.

```
(assert (= cs_001_x_001 2))
(assert (= cs_001_y_001 1))
```

Since we are running the program symbolically, we do not know which path is being followed. If we want to examine the path that skips the IF branch, we get as a condition

```
(assert (not (< cs_001_x_001 cs_001_y_001) ))
```

For this condition, the SMT solver will find that it is not true. For the path that does follow the IF branch, we get the following conditions.

```
(assert (> cs_001_x_001 cs_001_y_001))
(assert (= cs_001_x_002 (+ cs_001_x_001 cs_001_y_001)))
(assert (= cs_001_y_002 (- cs_001_x_002 cs_001_y_001)))
(assert (= cs_001_x_003 (- cs_001_x_002 cs_001_y_002)))
```

If a function calls another function, we also add the name of that function with a function counter to the name. This also happens when a function calls itself. If the function `f` calls the function `g`, the variable `a` in the function `g` is named:

```
$f_functioncounter2_g_functioncounter1_variablename_varcounter
```

5.5.2. FUNCTION RETURN VALUES

When a function returns values, there is generally an assignment of that return value to a variable. We can think of these return values as local variables. With return values it is necessary to give these values a unique name, because a function may contain multiple return statements. The naming scheme is the same as for normal variables, except that for the return value a new variable name is introduced (`ret`):

```
$function_functioncounter_ret_returncounter
```

On the first call of the function `f`, the following variable name is used on the second return in that function.

```
$f_001_ret_002$
```

BUILT-IN FUNCTIONS

PHP has a large number (about 900) of built-in functions. A list of the functions can be retrieved with the code in listing 5.2.

```
1 <?php
2 $arr = get_defined_functions();
3 print_r($arr);
4 ?>
```

Listing 5.2: PHP code to retrieve all built-in functions

If we want to include these functions in symbolic code, we need to know how these functions change the state of the symbolic store. Functions with no return value are a complication. The function is executed for a reason. To know how the state of the program changes, we need a model that shows what the state can change and how that state change can affect elsewhere in the program. Without a return value, this is difficult to estimate.

If a library function does return a there return value, we can try to include the state change in the SMTLIB program.

5.5.3. STRING FUNCTIONS

Appendix D contains an overview of the String functions that SMTLIB can handle natively. For these functions a direct translation of a PHP function is possible. This is a small number of the large set of string functions offered by PHP ⁶.

There are examples of String functions that are translatable to SMTLIB.

⁶<https://www.php.net/manual/en/ref.strings.php>

NL2BR

The function `nl2br`⁷ puts an `
` in front of all instances of `\r \n`, `\n\r`, `\n` and `\r`. In the case of `\r\n` it is replaced by `
\r\n`. In a similar way, we can replace the other new line occurrences.

```
(declare-const s_001 String)
(declare-const s_002 String)
(declare-const s_003 String)
(declare-const s_004 String)
(declare-const s_005 String)

; s_001 is the original string
(assert s_002 (str.replace s_001 "\r\n" "<br>\r\n"))
(assert s_003 (str.replace s_002 "\n\r" "<br>\n\r"))
(assert s_004 (str.replace s_003 "\n" "<br>\n"))
(assert s_005 (str.replace s_004 "\r" "<br>\r"))
; s_005 contains the resulting string
```

SUBSTR_COUNT

The function `substr_count` counts how many times a substring occurs in a given string. If the substring does not occur, the return value is -1. The function `substr_count` is an example of a string function that can be constructed for a concrete number of occurrences of the substring. We can enforce that the string occurs a concrete number of times by requiring that each instance come before the next instance. After the last instance, we require that the substring not occur again. We can do that by using the SMTLIB function `str.indexof`.

```
(set-option :print-success false)
(set-option :produce-models true)

(declare-const s_001 String)
(declare-const substr String)
(declare-const index1 Int)
(declare-const index2 Int)
(declare-const index3 Int)
(declare-const index4 Int)
(declare-const index5 Int)
;(declare-const index6 Int)
;(declare-const index7 Int)

;(assert (= s_001 "babbab"))
(assert (= substr "a"))

; s_001 is the string to search in
; we assert that the substring occurs 2 times
; the index of the last assertion is -1
(assert (= index1 (str.indexof s_001 substr 0)))
```

⁷<https://www.php.net/manual/en/function.nl2br.php>

```

(assert (= index3 (str.indexof s_001 substr index2)))
(assert (= index5 (str.indexof s_001 substr index4)))
;(assert (= index7 (str.indexof s_001 substr index6)))

; the occurrences do not overlap
; the last occurrence does not exist
(assert (> index2 (+ index1 (str.len substr ))))
; index1 + len(substr)<=len(s_001)
(assert (<= (+ index1 (str.len substr)) (str.len s_001)))

(assert (> index4 (+ index3 (str.len substr ))))
(assert (<= (+ index3 (str.len substr)) (str.len s_001)))
;(assert (> index6 (+ index5 (str.len substr ))))

(assert (distinct index1 -1 ))
(assert (distinct index3 -1 ))
(assert (= index5 -1 ))

```

If we do assert nothing about s_000 (s_001 has an symbolic value) and substr="a", Z3 returns the value s_001="a a" in the model. If we do assert nothing about substr (substr has an symbolic value) and s_001="babbab". Z3 gives "ba" as a possible value for substr. Both are correct solutions.

5.5.4. ARRAY FUNCTIONS

PHP has a whole set of array functions⁸. The array function array_count⁹ is an example of a function for which we have not found a solution as a translation to SMTLIB. We review examples of functions with a possible translation to SMTLIB below.

IN_ARRAY

The function in_array() can be seen as a search function that searches for an item in an array. In the code for a similar search function in SMTLIB, we set a value for the element in the last line, but leave the index open.

```

(declare-const key Int)
(declare-const value Int)
(declare-const arr (Array Int Int))

; fill the array
(assert (= arr
  (store (store (store ((as const(Array Int Int)) 0) 0 1) 1 2) 2 3) ))

;we are about to search for the value 7
(assert (= value 7))

```

⁸<https://www.php.net/manual/en/function.array>

⁹<https://www.php.net/manual/en/function.array-count-values.php>

```
; the key for value 7 is
(assert (= value (select arr key)))
```

In this case, Z3 returns unsat because the value 7 does not exist in the array.

SORT

All conditions in SMTLIB must describe the change of state that has occurred by executing the corresponding PHP statement. For sorting, the following logic condition applies for elements of an array:

$$\forall_{i,j} i \leq j \implies a[i] \leq a[j]$$

We can also add these conditions to a SMTLIB program. We thereby require that the array elements be sorted.

```
; declaration of arrays
(declare-const arr (Array Int Int))
(declare-const arr2 (Array Int Int))
; indexes
(declare-const i Int)
(declare-const j Int)

; values to put in array at index 0,1,2
(declare-const in0 Int)
(declare-const in1 Int)
(declare-const in2 Int)

; values to be read from index 0,1,2
(declare-const out0 Int)
(declare-const out1 Int)
(declare-const out2 Int)

; store 3 symbolic values in array arr
; the resulting array is arr2
(assert (= arr
  (store
    (store
      (store ((as const(Array Int Int)) 0) 0 0)
      0 0)
    0 0) ))

(assert (= arr2 (store (store (store arr 0 in1) 1 in2) 2 in3) ))
;assert that the stored values are distinct
(assert (distinct in0 in1 in2))

; assert that arr2 is sorted
(assert (= true (forall ((i Int) (j Int)) (=> (<= i j) (<= (select arr2 i) (select arr2 j))))))

; retrieve the values at index 1, 2 and 3
```



```
(assert (= out1 (select arr2 0) ))
(assert (= out2 (select arr2 1) ))
(assert (= out3 (select arr2 2) ))
```

FLOOR

Some PHP functions can be defined in SMTLIB by using functions that are already present in SMTLIB. An example is the `floor()` function which rounds a float down:

```
(define-fun floor ((y Real)) Int (to_int y))
```

SYMBOLIC RETURN VALUES (E.G. RANDOM_INT)

There are several possibilities to include the return value of a library function in the SMTLIB program. As an example we will look at the `random_int` function of PHP.

The simplest possibility is to choose a random concrete function value for the random function. In general, we cannot say what influence this has on the degree to which the program can be executed symbolically. We have thus introduced an element of concholic testing.

A second possibility is that we choose a symbolic function value for random function. In addition, we can specify a range¹⁰.

```
(declare-const random_001_ret Int)
(assert (>= random_001_ret 10 ))
(assert (<= random_001_ret 20 ))
```

A final option is to execute the operation random function completely symbolically. This is possible in principle because the source code is available¹¹. However, the goal is not to find vulnerabilities in the random number generator, so we suffice with the assumption that with a symbolic output we can take the function along well enough.

¹⁰<https://www.php.net/manual/en/function.rand.php>

¹¹https://github.com/php/php-src/blob/master/ext/standard/mt_rand.c

5.6. TYPE INFERENCE

Before the PHP statements in a path can be translated to SMTLIB, we need to know what type the variables are. This is necessary because SMTLIB is a statically typed language. If we execute a program with concrete values, a variable at some point in the code flow has a certain value and the corresponding type. With symbolic execution, the value is free. Therefore, several interpretations of the code are possible. This also means that if several types are possible for a PHP expression, several different translations to SMTLIB are also required. Furthermore, the type and the operator to be used are related. For example, the type determines which comparison operator can be used in SMTLIB (See Appendix D). For the translation to SMTLIB, it must be clear which type is involved when a change of state occurs. Van der Weijde [53] studied constraint based type inference for PHP. We have adapted the type inference system discussed in van der Weijde's [53] master thesis for our purpose. Furthermore, we found inspiration in the master thesis of Klingstrom and Olssen [30], who investigated whether Deep Learning can be used for Type Inference in PHP. We do not need to use the algorithm discussed in [1], as we are only looking at cycle free paths and each variable is only assigned a value once (SSA). We suggest a simple algorithm that infers the type of all expressions.

Van der Weijde's type system assumes a type tree for PHP (Figure 5.2). When we concretely run the program, an expression has one specific type. using symbolic expression, the type of an expression can be a set of types.

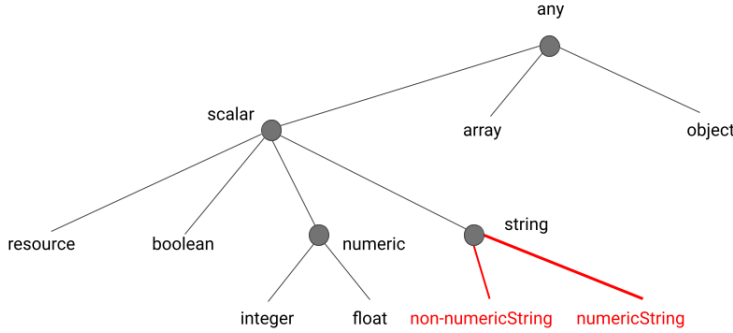


Figure 5.2: adapted PHP type tree

Since we also want to describe Type Juggling with our type system, we add the type `numericString`. A numeric String is a string that starts with a string representation of a numeric possibly followed by other characters. A numeric string is converted to a numeric by Type Juggling. The second adaption of van der Weijde's system part of the system consists of derivation rules for the various operators. A rule of inference gives the type of expression that results from the operator. The rules are listed as

$$\frac{\text{premise 1} \quad \text{premise 2}}{\text{constraint 1} \quad \text{constraint 2}}$$

Van der Weijde writes the inference rule for the greater than operator as:

$$\frac{E \equiv (E_1 > E_2)}{[[E]] = \{boolean\}}$$

We add extra type rules for premises of constraints. For the greater than operator out adapter version is:

$$\frac{[[E_1]] = \{integer\} \quad [[E_2]] = \{integer\} \quad [E \equiv (E_1 > E_2)]}{[[E]] = \{boolean\}}$$

The inference rule can be explained as follows. In the top right-hand corner of the line is the expression to which the inference rule applies. We see that in the case of the greater than operator, it compares two expressions E_1 and E_2 . The type that the expression E_1 and E_2 must satisfy is shown in the top left-hand corner of the line. Below the line is the type that results from the greater than operator. A comparison operator does not only work on integers. We can also create an inference rule for when the two expressions are of type string.

$$\frac{[[E_1]] = \{string\} \quad [[E_2]] = \{string\} \quad [E \equiv (E_1 > E_2)]}{[[E]] = \{boolean\}}$$

Thus, for an operator there may be more than one inference rule. Each rule applies to a specific combination of types of expressions.

If we want to apply an inference rule, we first search for the inference rules that apply to the operator in question. A partial set of inference rules for PHP can be found on github ¹².

¹²<https://github.com/rdohmen/grad>

As an example we will now determine the type of the variables in listing 5.3.

```
1 $b=456;
2 $c="123string";
3 $a=(b<c);
```

Listing 5.3: PHP program to illustrate type inference

For the first rule we need an inference rule for an assignment.

$$\frac{[[E_2]] = \{integer\} \quad E \equiv (E_1 := E_2)}{[[E_1]] = \{integer\}}$$

Van der Weijde provides a rule with subtypes for this. We choose to specifically mention the allowed type. The inference rule below for an assignment of type numeric is interesting at first glance because it can be used to summarise a number of other inference rules. The problem in our context is that the type numeric leaves doubt between the PHP types integer or float (or in the case of smtlib between integer and real).

$$\frac{[[E_2]] = \{numeric\} \quad E \equiv (E_1 := E_2)}{[[E_1]] = \{numeric\}}$$

We can use the inference rule because it is a rule for an assignment and because the right-hand side is an expression of type integer. There are two inference rules that can be used in this case: the inference rule for assignment with a string and a similar one for a numericString.

$$\frac{[[E_2]] = \{string\} \quad E \equiv (E_1 := E_2)}{[[E_1]] = \{string\}}$$

$$\frac{[[E_2]] = \{numericString\} \quad E \equiv (E_1 := E_2)}{[[E_1]] = \{numericString\}}$$

We conclude that there are two possibilities. c is of type string or of type numericString.

$$[[c]] = \{String, numericString\}$$

Before we can determine the type of a, we must first determine the type of the equation. For both cases there is an inference rule.

$$\frac{[[E_1]] = \{integer\} \quad [[E_2]] = \{string\} \quad [E \equiv (E_1 > E_2)]}{[[E]] = \{boolean\}}$$

$$\frac{[[E_1]] = \{integer\} \quad [[E_2]] = \{numericString\} \quad [E \equiv (E_1 > E_2)]}{[[E]] = \{boolean\}}$$

So in both cases (c is a String or a NumericString) we can determine that the type of a is Bool.

Data: Given: Path P, Set of Type Inference Rules R

TS_Tree.create(P.firstNode);

Q.add(P.FirstNode);

while $Q \neq \emptyset$ **do**

$Q \leftarrow \emptyset$;

forall $q_i \in Q$ **do**

$q_i = Q.pop()$;

$op \leftarrow q_i.operator$;

$ExprList \leftarrow q_i.ExprList$;

forall $r_i \in R$ **do**

if $R.match(r_i, op, ExprList)$ **then**

$new_node \leftarrow R.apply(r_i, q_i)$;

 TS_Tree.add(q_i, new_node);

$Q'.add(new_node)$;

end

end

end

$Q \leftarrow Q'$;

end

Algorithm 2: A Type Inference Algorithm

<i>PHP</i>	<i>Code</i>	<i>SMTLIB</i>
<pre> <?php function maximum(\$a,\$b) { if(\$a>\$b) { \$m=\$a; } else { \$m=\$b; } return \$m; } function minimum(\$a,\$b) { if(\$a<\$b) { \$m=\$a; } else { \$m=\$b; } return \$m; } function add(\$a,\$b) { \$add=\$a+\$b; return \$add; } for(\$i=maximum(2,3); \$i=5; \$i=add(\$i,1)) { echo(\$i); } ?> </pre>	<pre> max_a>max_b max_m=max_a max_m=max_b res001=max_m add_add=add_a+add_b res002 = add_add max_a=2 max_b=3 execution of max main_i=res001 main_i=5 add_a=i add_b=1 execution of add res003=main_i execution of echo </pre>	<pre> (assert (> max_a max_b)) (assert (= max_m max_a)) (assert (= max_m max_b)) (assert (= res001 max_m)) (assert (= add_add (+ add_a add_b))) (assert (= res001 add_add)) (assert(= max_a 2)) (assert(= max_b 3)) execution of max (assert(= i res001)) (assert (= main_i 5)) (assert(= add_a main_i)) (assert(= add_a 1)) execution of add (assert(= res003 main_i)) execution of echo </pre>

Table 5.3: Example of a translation of PHP code including a function to SMTLIB

6

SOLVING PATH CONDITIONS

6.1. DRAW UP PATH CONDITIONS

When we get to the point of solving path conditions, we have gone through a number of steps.

- We have converted the PHP code into an ICFG
- In the ICFG, we searched for paths that met certain conditions.
- For these paths we made a translation to SMTLIB-statements as complete as possible.

For each path, a list of SMTLIB statements was drawn up. We call this list of conditions a program in this section. The program consists of conditions that together describe all state changes of the symbolic store that occur when the PHP statements in a concrete path are executed symbolically.

An example of such a program can be found in listing 6.1 (this example is an translation of one of the Samate Samples).

```
1 (set-option :print-success false)
2 (set-option :produce-models true)
3 (declare-fun A () (Array Int String))
4 (assert (= A (store A 0 "safe")))
5 (declare-const get_user_data String)
6 (assert (= A (store A 1 get_user_data)))
7 (assert (= A (store A 2 "safe")))
8 (declare-const tainted String)
9 (assert (= (select A 1) tainted))
10 (declare-const tainted2 Int)
11 (assert (= tainted2 (str.to.int tainted)))
12 (assert (> tainted2 0))
13 (declare-const res Int)
14 (assert (= res tainted2))
```

Listing 6.1: Example of PHP code that allows SQL injection, because the input has not been sanitised

6.2. SOLVING CONSTRAINTS WITH Z3

We now want to let an SMT solver determine a solution for the path conditions for a path. This can be done by calling the SMT solver from within a program via an API.

The developers of Z3 mention a number of API links to programming languages ¹. We have rewritten an unofficial link to Delphi (Z34Delphi ²) to Lazarus (an Open Source clone of Delphi): Z34Lazarus ³. The goal was to use this link to loop the SMTLIB scripts into Z3 without a diversion. Unfortunately, the SMTLIB link turned out not to work properly, so another solution had to be found.

Therefore, we have written the python program `exec_smt.py` to execute the SMTLIB programs. The python program expects the name of a file containing an smtlib program as a parameter.

```
1  #!/usr/bin/env python
2  import sys
3  from z3 import *
4
5  with open(sys.argv[1], 'r') as my_file:
6      cmd = my_file.read()
7      print(cmd)
8
9      s = Solver()
10     s.from_string( cmd )
11
12     result = s.check()
13     print(result)
14
15     with open(sys.argv[1]+".sat", "w") as file_object:
16         # (over)write the result (sat|unsat|unknown) to the file
17         file_object.write(str(result)+"\n")
18
19     #append result to file with results
20     with open("results.txt", "a") as file_object:
21         # Append the result (sat|unsat|unknown) at the end of file
22         line=sys.argv[1]+" "+str(result)+"\n"
23         file_object.write(line)
24
25     #write a model in case the conditions are sat
26     if str(result)=="sat":
27         model=s.model()
28         print(model)
29         with open(sys.argv[1]+".model", "w") as file_object:
30
31             file_object.write(str(model))
```

¹<https://github.com/Z3Prover/z3>

²<https://github.com/Pigrecos/Z34Delphi>

³<https://github.com/rdohmen/Z34Lazarus>

Listing 6.2: Example of PHP code that allows SQL injection, because the input has not been sanitised

After installing the Z3 module

```
pip install z3-solver
```

a SMTLIB program in the example.smt file, can be executed as follows

```
python exec\_smt.py example.smt
```

The program saves the result in three ways:

- A file example.smt.sat is created in which the result (sat, unsat, unknown) is stored.
- In the file result.txt a line is added with the file name and the result (example.smt sat)
- In the case of a fulfilling set of conditions, a model is stored in the example.smt.model file.

The model is one of the possible solutions of the SMTLIB program. We can only influence the chosen model by adding extra conditions (which are independent of the PHP program itself).

6.3. SAMATE

One of the research goals was to see to what extent we could symbolically execute the safe and unsafe PHP samples in the SAMATE dataset.

Because the development of our tool is not yet so advanced that it can produce the SMTLIB code for a path, we have chosen to analyse samples manually.

There are a number of viewpoints associated with this question.

- The number of samples in the data set.
- The complexity of the individual samples.
- The extent to which the samples can be executed symbolically.
- The degree to which the answers of the SMT solver really say something about the (in)security of the samples, but also the degree to which the samples are really usable as examples of (in)secure code.

6.3.1. THE SIZE AND COMPLEXITY OF THE DATASET

The samples of the SAMATE set are assembled according to a certain pattern [49]. Each code fragment starts with a part where input enters the code. In the second part, the input is sanitised. Finally, sanitised input enters a sink where the input is used in output.

The total number of samples is large, because a limited number of variants for the input, the cleaning and the sink are combined. Because the samples are composed in this structured way, we do not need to study all the samples to be able to assess them.

The cyclomatic complexity [36] of the samples is low. Often no conditional statement is included in the code. In the most complex samples, an IF and a WHILE statement are present. Thus, the CFGs of these programs are often very simple. In many cases there are

one or two paths, only if a WHILE is present potentially infinite paths are possible. This WHILE loop is always used for code that needs to check the rules of a database query. It does not matter for the vulnerability how often the loop is executed. Since the operation of the database function cannot be verified with symbolic execution, it is defensible to assume that the vulnerability occurs on the first pass.

6.3.2. USABILITY

We look at two examples to see to what extent the sample can be used as an example of safe and unsafe code.

EXAMPLE 1

In the filename⁴ the sanitize part is "CAST-cast_float_sort_of". This is an example of a safe sample. The description in the file name suggests that a cast to a float occurs and that something is going to happen with sort, of that it might be "a sort of a cast". We recognize the cast in line 6 of the code in listing 6.3.

```

1 $array = array();
2 $array[] = 'safe';
3 $array[] = $_GET['userData'];
4 $array[] = 'safe';
5 $tainted = $array[1];
6
7 $tainted += 0.0;
8
9 $query = "find / size '". $tainted . "'";
10
11 $ret = system($query);

```

Listing 6.3: Example of PHP code in which an implicit cast is used to enforce that the input is a numerical value

The type of `$_GET` is String. Adding a Float to a string is probably to enforce a numeric value. As of PHP 7 (it is not exactly clear which version), this command gives the warning: "WARNING: non-numeric value encountered. This warning is comprehensible, because the addition statement implicitly performs a cast, by applying Type Juggling. If the string in `$array[1]` starts with a numeric value, this numerical value is added to 0.0. The decimal part is omitted if it is equal to 0.

value of \$tainted	value of \$query after implicit cast
\$tainted = "string";	\$query = "find / size 0"
\$tainted = "2string";	\$query = "find / size 2"
\$tainted = "2.5string";	\$query = "find / size 2.5"

Table 6.1: Examples of how line 6 sanitizes the input

The query contains the text "find / size". It is not clear how this is meant. It may indicate that both the command FIND and the command SIZE can be used. Under Windows and

⁴CWE_78__array-GET__CAST-cast_float_sort_of__find_size-concatenation_simple_quote.php

Linux, the FIND command has no parameter /size . Moreover, there must be a directory name between the find and the parameter. This can be a /. A minus sign must precede the parameter size (-size).

Probably the intention is to search for files of a certain size: FIND / SIZE <numeric value>. Looking at the possibility of exploiting a vulnerability here, the obvious thing to do would be to prevent text from being entered after the -size parameter other than the numeric value. With an && operator, another command could be added after the FIND command. By the way, the numeric value in the string \$query is as between two apostrophes. That already ensures that a command added with && will not work.

Finally, according to the manual, the numerical values can be preceded by +- and succeeded by one of the characters 'cbwkMG'. Casting to a number filters these characters from the input.

So there are quite a few snags in this example. The example probably does not do exactly what was intended. One has to guess what the exact intention is and the question is whether all the details were really intended that way.

EXAMPLE 2

In the filename ⁵ the sanitize part is "FILTER_CLEANING-FULL_SPECIAL_CHARS". The source code of this unsafe sample is shown in code in listing 6.4. The built-in PHP function filter_var() is used to sanitize the input.

```
1 $array = array();
2 $array[] = 'safe';
3 $array[] = $_GET['userData'];
4 $array[] = 'safe';
5 $tainted = $array[1];
6
7 $sanitized = filter_var($tainted, FILTER_SANITIZE_FULL_SPECIAL_CHARS);
8 $tainted = $sanitized;
9
10 $query = "ls '". $tainted . "'";
11
12 //flaw
13 $ret = system($query);
```

Listing 6.4: Example of PHP code that allows SQL injection, because the input has not been sanitised

In the first five lines of the sample, an array is filled. The second element of the array is populated with the input of a \$_GET. This input is filtered for special HTML characters in line 7. The character &, single and double quotes, < and > are replaced by their HTML equivalents. This filter thus prevents these characters from ending up in the query string of line 10. A concatenation with && will therefore no longer appear in the string. The > is also replaced. Characters such as | and (remain in the string, however, so a concatenation with || remains possible. It is therefore still possible to add a second command after the query.⁶

It is not immediately obvious from the code fragment why it is an unsafe sample. It is only clear once we know that the filter_var function passes symbols that make it possible

⁵CWE_78__array-GET__func_FILTER-CLEANING-full_special_chars_filter__ls-concatenation_simple_quote.php

⁶<https://www.man7.org/linux/man-pages/man1/bash.1.html>

to build a query with malicious code

6.3.3. CODE COVERAGE

We could consider code coverage as a criterion to determine how well we can use symbolic execution to execute a PHP program. The problem here is that a single command that cannot be executed symbolically can lead to an unusable solution. The weakest link can cause the whole system to collapse. Therefore, we think that a percentage does not clarify to what extent symbolic execution is usable. It is better to get a concrete picture of the kind of problems that can be solved or are impossible to solve with symbolic execution.

BEGIN BLOCKS

The PHP samples begin with a choice of a limited number of code blocks ⁷. Non of these begin blocks contain PHP code.

INPUT

For the input, the code generator can choose from 16 different code fragments. An overview of these fragments can be found on github⁸. We have described a possible translation of these fragments to SMTLIB. In addition, we have looked at for which specific language fragments the tool should be able to generate SMTLIB code:

- A: Determine result of built-in function
- B: null type return value
- C: type of by reference parameter
- D: retrieve array element by index
- E: resource type return value and type juggling
- F: Two dimensional array
- G: Classes (create instance, execute constructor, execute method)
- H: Automatic indexing for arrays

example

For the following code fragment the tools should be able to generates code code that determine result of built-in function, determines the type of a by reference parameter and retrieve an array element by index.

```
$script = "/tmp/tainted.php";  
exec($script, $result, $return);  
$tainted = $result[0];
```

If the tool has these three capabilities, 12 of the 16 fragments can be translated. If the tool is able to translate all eight requirements, all fragments can be translated. For other combinations of the eight possible requirements, these numbers are different. For example, if we want to be able to use all fragments with a class, the tool must also be able to

⁷<https://github.com/stivalet/PHP-Vuln-test-suite-generator/tree/master/bin/fileSample>

⁸<https://github.com/rdohmen/grad>

retrieve an array element by index and must be able to automatically index newly added array elements.

Determine result of built-in function We saw earlier using the example `rand_int` that for the return type of a built-in function we can assume a symbolic variable of the appropriate type, supplemented by additional conditions that limit the possible values. In many other cases, a similar translation is possible.

null type return value The PHP type `null` does not exist in SMTLIB. The solution that ignores these situations in which a null type can occur is not satisfactory. With null types, often no other operations are performed as (implicit) cast to a Boolean for conditional statements. A translator could generate two type paths for `True` and `False` in this case.

type of by reference parameter There are functions in PHP that use a by reference variable to return a return value. We have not included by reference variable in our design. If they are used not only for normal variable, but also for by reference parameters, they can be handled the same way as symbolic return values.

retrieve array element by index We used the `in_array` function examples to show how this language construct can be translated into SMTLIB.

resource type return value and type juggling We have not studied the PHP type of resource (which does not appear in SMTLIB).

Two dimensional array We have not studied more dimensional arrays, but we expect that it is possible to use this type if the type of a dimension is fixed.

Classes (create instance, execute constructor, execute method) We have not worked out classes due to lack of time. For simplicity, we could see a class as a collection of variables and functions that operate on these variables. With inheritance added, a complex situation arises. Nevertheless, we expect this to be promising since the problem is built from parts already studied.

Automatic indexing for arrays

If the index values are concrete, the translator can keep track of which indexes are used. For the next index the highest value plus one is taken. If symbolic values for the indexes are used in the meantime, a complex situation arises in which it is no longer clear which indexes are free.

```
$array = array(1, 1, 1); // index 0,1,2
$array[5]=1;           // index 5
$array[S]=1;           // create a new index, append index 6,
                        //or overwrite an existing index
```

SANITIZING THE INPUT

For the sanitising methods that appear in the SAMATE samples, we have looked at the extent to which they can be executed using symbolic execution.

There are 42 code fragments that can be divided into 18 categories.

- A crypt
- B preg_match
- C escapeshellarg
- D filter_var
- E cast to float
- F type juggling
- G cast to int
- H issettype
- I addslashes
- J htmlentities
- K htmlspecialchars
- L mysql_real_escape_string
- M preg_replace
- N ternary operator
- O in_array
- P esapi
- Q pre_replace
- R array

Determine result of built-in function The function crypt is an example like the functions we have seen before, for which for the return value a symbolic string value can be taken, possibly supplemented by additional conditions to constrain the solutions.

preg_match, pre_replace These functions are examples of symbolic execution with regular expressions. We will see in the examples of paragraph 6.4.3 that Z3 cannot reason with symbolic expressions.

filter_var The function filter_var is a complex function that can take many different forms. These functions can be grouped into two categories with a few exceptions (callbacks): sanitize and validate filters. The sanitize filter FILTER_SANITIZE_EMAIL is a function that removes all characters except letters, numbers, and !#\$%&'*+,-=?_@.[]|. This function can be recreated with str.replace. We know from earlier examples we examined that Z3 can reason symbolically with strings. The validate filter FILTER_VALIDATE_EMAIL checks whether a string is in the form of a valid email address using a regular expression. We know from the examples examined in paragraph 6.4.3 that Z3 cannot reason with symbolic regular expressions.

esapi Esapi⁹ is a security framework that provides functions via an API. We have not reviewed this framework.

cast to float, cast to int For casts to integer and float values, SMTLIB has built-in functions.

type juggling Type juggling is possible because we consider the numericType separately. We can require that the string is composed of a number followed by a non-numeric string.

array, in_array We have described the automatic indexing of array elements above

ternary operator We have not yet looked at this operator, but we expect that it can be incorporated into the methodology in a similar way.

mysql_real_escape_string The function mysql_real_escape_string is an example of a function that can be recreated with str.replace.

escapeshellarg The function escapeshellarg is an example of the system functions¹⁰ that return a string as a return value. These functions, like others that return a string, can be recreated by introducing a symbolic string variable with additional limiting conditions.

isset, unset The isset function is used to verify that a variable exists (is not null). A variable can be made null with the function unset. These functions should make it possible to work with the null type.

addslashes, htmlentities, htmlspecialchars The function addslashes places a slash before a single quote ('), double quote ("), backslash and NUL (the ZERO byte). Except for the last character, this should be achievable in SMTLIB with str.place conditions. The function htmlentities is an alias of htmlspecialchars that replaces some character with an html equivalent. This function can also be implemented with str.place.

SINK

In the SAMATE samples, two systems are used to construct an exit point (the sink). On the one hand, a small selecting of possible end blocks (fixed code fragments) that terminate the sample. On the other hand, a selection of more extensive code fragments that can serve as an exit point.

END BLOCKS

The PHP samples are closed with a choice of a limited number of code blocks¹¹. The closing code sometimes includes a sink. If that is the case, it is always an echo statement containing the \$tainted variable.

OTHER CODE FRAGMENTS

The remaining code snippets consist of 134 examples where a variable is used in a command that we can think of as an exit point. These commands fall into 18 categories::

⁹<https://github.com/OWASP/owasp-esapi-php/blob/master/src/ESAPI.php>

¹⁰<https://www.php.net/manual/en/function.escapeshellarg>

¹¹<https://github.com/stivalet/PHP-Vuln-test-suite-generator/tree/master/bin/fileSample>

- A Try except
- B eval
- C echo
- D conn→prepare
- E mysql_query used as exit point
- F system
- G ldap_search
- H str_rot13
- I sha1
- J crypt
- K password_hash
- L sprintf
- M fopen
- N header
- O http_redirect
- P include
- Q require
- R string containing variable as a literal

Only 1 code fragment contains a WHILE loop. Two code snippets contain an IF statement. As with the previous sections (input and sanitize), the tool must be able to process the statements. It is not necessary to further process the variable that is passed as a parameter to the function.

Function with a symbolic output value This symbolic output value is then displayed with an echo. This applies to the functions: str_rot13, sha1, crypt, password_hash.

variable passes (malicious) input to a function For most fragments, the malicious output is passed to the function via a variable. Examples are eval, echo, conn=>prepare, mysql_query, system, ldap_search, system, ldap_search, sprintf, fopen, header, http_redirect, include, require, string containing variable as a literal.

Try except We have not fully studied this language element. To include try catch blocks correctly it must be known which exception a statement/function can raise. This exception is then handled outside the function that raised the exception. The code flow in general is probably complicated.

6.4. SPECIFIC CODE FRAGMENTS

In this section we will look at specific code fragments to see to what extent SMTLIB can simulate the behaviour of PHP.

6.4.1. REGULAR EXPRESSIONS

It is possible in SMTLIB to execute conditions relating to regular expressions. For example, it is possible to check whether a concrete string matches a regular expression. However, we want to match a symbolic string with a concrete regular expression. We can write code in smtlib for this purpose.

```
(set-option :print-success false)
(set-option :produce-models true)

;$array = array();
;$array[] = 'safe' ;
(declare-const array0 String)
(assert (= array0 "safe"))

;$array[] = $_GET['userData'] ;
(declare-const array1 String)
(declare-const get_userdata String)
(assert (= array1 get_userdata))

;$array[] = 'safe' ;
(declare-const array2 String)
(assert (= array2 "safe"))

;$tainted = $array[1] ;
(declare-const tainted String)
(assert (= array1 tainted))

;$re = "/^[a-zA-Z]*$/";
(declare-const regex RegLan)
;(declare-const re String)
;(assert (= re "/^[a-zA-Z]*$/"))
;(assert (str.to_re re regex) )
(assert (= regex (str.to_re "/^[a-zA-Z]*$/")))

;if(preg_match($re, $tainted) == 1){
; RE membership
(declare-const result Bool)

(assert (= (str.in_re tainted regex) true))

;(declare-const len Int)
(assert (> (str.len get_userdata) 3))

; $tainted = $tainted;
;}
;else{
; $tainted = "";
```

```

;}
;$query = "ls '". $tainted . "'";
;$ret = system($query);

```

Unfortunately, Z3 does not generate an instance of a String that matches the regular expression in the model. Z3 returns the same regular expression as a solution. That is a correct, but unusable solution.

6.4.2. STRING FUNCTIONS

We will review to what extent string functions can be incorporated into an SMTLIB program.

The program haystack.smt (See listing 6.5) examines whether a string exists that is a substring of a given string. The given string is called haystack and contains the string "aneedleinah".

```

1 (declare-const haystack String)
2 (declare-const needle String)
3 (assert (= haystack "aneedleinah"))
4 (assert (> (str.indexof haystack needle) -1))

```

Listing 6.5: Example of PHP code that allows SQL injection, because the input has not been sanitised

The program is sat and in the model the value of the string variable needle is a substring of the variable haystack. When executing the program, the value of needle is not always the same. We got the values "eedleina" or "aneedlei". The latter solution is more obvious, because a string that corresponds to the first n characters of the string being searched is always a substring. But apparently Z3 also searches for less obvious substring sometimes.

6.4.3. NUMERIC STRINGS

With Type Interference, strings representing an integer or float are cast to that number in certain operations. We look at whether we can simulate this behaviour with symbolic execution.

If we use a concrete string s, Z3 manages to find a solution. We take the first character of the string and condition it on an integer greater than 0 (-1 is an error code for situations where the string to be converted contains integer representation).

```

(assert (= s "3string"))
(assert (= a (str.substr s 0 1) ))
(assert (= i (str.to_int a) ))
(assert (> i -1))

```

Z3 returns the solution: sat, with model: [s="3string", a="3", i=1].

We will now see if Z3 can also determine a solution for a symbolic string. We give as constraint for the string s that it must contain at least two characters (otherwise this gives Z3 the freedom to come up with a minimal solution for the string containing only a number). We use the function str.isdigit to enforce that the first character is a digit:

```

(assert (> (str.len s) 1))

```

```

(assert (= a (str.substr s 0 1) ))
(assert (= b (str.is_digit a) ))
(assert (= b true))

```

Z3 gives the solution: unknown. If we check with the str.to_int cast, Z3 does find a solution.

```

(assert (> (str.len s) 1))
(assert (= a (str.substr s 0 1) ))
(assert (= i (str.to_int a) ))
(assert (> i -1))

```

Z3 gives as solution: sat. With model: [s="0 ", a="0", i=0]. The string "0 " given by Z3 is a correct string (the second position contains a space). We can try to generate a more realistic example if we require the second character of the string s not to be a space. Therefore we add the assertion (assert (distinct " " (str.substr s 1 1))). The solver then returns "unknown" as a solution.

7

CONCLUSION

7.1. DISCUSSION

In this section, as the first part of the conclusion, we discuss the results of the study per research question.

7.1.1. RQ1: HOW CAN WE FIND PATHS IN PHP-CODE EFFICIENTLY?

We will discuss the result of the first research question through the four sub-questions.

The first step, creating an AST from a PHP program, was easy, because PHP-ast uses an extension that has made it possible to export the AST of a PHP program since PHP 7. Converting most conditional statements into an AST was successful, so in the CFGs we can create, the paths between conditional statements are fully modeled for most programs. Only the FOREACH statement has not been worked out. The Continue and BREAK statement is also well modeled. A special feature of these statements is, that they can not only jump out of the statements (SWITCH, DO-WHILE, WHILE, FOR) they are used in, but they can also jump back several levels within nested loops. Handling user-defined functions is done by inserting the CFGs of the functions where there is a function call. Of the built-in functions, only the symbolic return values are used. This can sometimes have complicated consequences, because it must be guaranteed that symbolic return value matches state (think of the count function that counts the number of elements in an array).

It is possible in PHP to create function variables. If the function name has a concrete value, the CFG of the function is inserted as for normal function calls. For the situation where the function name has a symbolic value, we have found some ideas for partial solutions, but no conclusive solution that works in all situations. Classes are not included in the tool, but we believe that to the extent that we store Classes in separate variables and can think of these Classes as a collection of variables and functions that act on those variables, it is likely that they can be included in a CFG in a similar way.

The proposed algorithm that searches paths in a CFG (Algorithm B) can find paths between nodes throughout the search space. We can easily limit the search space by specifying a goal that the paths must satisfy. We can do this by specifying how often each node may appear in the path, or what length the path may have. By setting the target frequency for all nodes to $\mathit{mbf}N$, the algorithm will accept all paths as solutions. Paths without cycles can be found by setting the target frequency for all nodes to 1. Prohibited regions can be specified by setting the target frequencies for the nodes in those regions to zero. In this

way, we can enforce that branches or loops are kept out of solutions, or enforce that solutions go through these branches or loops (and how often). In this way, we can even search in a limited way for paths that do or do not meet certain safety conditions. By choosing a target frequency of zero for the node after a WHILE loop, we can search for paths for which the condition of a WHILE loop is never false. This could happen in the case of a DoS attack where the program enters an eternal loop.

7.1.2. RQ2: UP TO WHAT EXTENT CAN PHP-CODE BE EVALUATED USING SYMBOLIC EXECUTION?

We have described a method that allows us to translate PHP code into SMTLIB code. It helps that Algorithm B only produces linear paths, because then a path consists only of consecutive statements. The SMTLIB code must describe the state change of the corresponding PHP statement. As far as the types and corresponding operators exist in both languages, the translation is possible one-to-one. For string functions, this is possible with a limited number of functions. We have shown examples of translations of PHP string function by using with the string functions provided by SMTLIB. For arrays, we face the limitation that arrays in SMTLIB have a fixed type for the indexes and values. Arrays with a mixed type cannot be translated to SMTLIB without much effort. It is possible to create an SMTLIB translation for a limited number of array functions. These include a search function and a sort function.

7.1.3. RQ3: UP TO WHAT EXTENT CAN PATH CONDITIONS BE SOLVED?

To what extent can the conditions be solved by an SMT solver? For the basic Int, Real and Bool and String types, Z3 often provides a suitable model when symbolic variables are used. However, there are examples where Z3 gives solutions that are not useful in relatively simple situations. This happens when Z3 must reason with symbolic regular expressions. Other research shows that it is possible to derive examples of regular expressions with SMT solvers other than Z3. In other situations where the conditions for a complex path need to be calculated, on the one hand, Z3 does come up with the correct solution, but the solution is not usable because although the solution falls into the search space, it does not resemble what a human user would have expected. This happens when Z3 needs to find solutions involving from symbolic strings. The solutions are then sometimes of an unusable form. Attempts to force Z3 to generate a real world solution sometimes end with no solution at all.

7.2. RESEARCH CONTRIBUTION

A number of results emerge from the study. First, we have shown how the can structure conditional statements of a PHP can be represented in a CFG. We have presented a search algorithm that can efficiently search for paths in a CFG by setting appropriate targets. For a path, we have proposed a type inference algorithm that can determine which type paths belong to a path, so that depending on the type of the variable, matching SMTLIB translations are made. For a small subset of PHP, we investigated the possibility of translating to SMTLIB conditions. It has become clear that this translation is not usable in some situations due to limitations of the Z3 SMTLIB solver.

7.3. LIMITATIONS AND FUTURE WORK

The result of this study has limitations in several areas of the approach taken.

When converting an AST to a CFG, we did not include several language elements of PHP. If these language elements occur in a code path of a PHP program, the incomplete translation to logical conditions may cause the solutions of an SMT solver to be completely different from the actual solutions. Extending the methodology to include these missing language elements may at least partially solve this problem. Examples are the ability to handle classes and reference variables.

The proposed search algorithm is a modification of known search algorithms that can flexibly influence which paths are considered solutions based on two properties (the number of times a node occurs in a path and the number of nodes in a path). The results of this part of the research can be used as a starting point for further research in several ways. First of all, it is difficult to prevent an arbitrarily chosen set of target frequencies from making any path impossible. There are situations that are easy to recognise, for example when in successive nodes in a part of a path without branches a node with a low target frequency is followed by a node with a higher target frequency. In addition, other targets can be included, these targets could be related to the type of statement that occurs in the path.

When translating PHP statement to SMTLIB, we quickly run into limitations of SMTLIB. Research could find possible translations for types and functions that do not exist in SMTLIB.

By adding safety conditions we might be able to find more meaningful solutions that trigger not only the path but also the vulnerability.

For a path that is executed symbolically, many solutions may exist. The set of input values (the model) that the SMT solver determines is one of the possible sets of solutions for which the path can be executed. If we want to look more specifically for paths that lead to vulnerabilities, we can look at the conditions under which the vulnerability is not caused. We call these conditions the security conditions (SC). If the path condition is met, but the security conditions are not met, then we have the conditions for a code path that leads to a vulnerability and also triggers this vulnerability.

We can only run a program from any language (such as PHP) symbolically in SMTLIB if we can translate the conditions that follow from statements from the source language into SMTLIB (the target language). We have described for a limited number of functions how this is possible, but there are many parts of PHP for which it is possible to investigate if and which translations into SMTLIB are possible.

7.4. RELATED WORK

In this section we discuss research that touches upon aspects of the proposed research. We look at aspects of finding paths in graphs, symbolic execution and testing tools that were the result of previous research.

In our research we combine techniques and frameworks of previous research. The use of graph-representations to analyze code, search algorithms to find paths and the use of SMT-solvers to solve for constraints resulting from Symbolic Execution has been used in previous research. Existing frameworks ¹ proved their usability and will therefore also be considered in our investigation.

7.4.1. FINDING PATHS IN A GRAPH

When looking for paths, we have to distinguish between looking for paths in a mathematical graph or looking for code paths. The graphs described in the problem analysis are directed graphs. ASTs are trees [51]. If we want to find a path starting from a node in a tree, we can search forward from the start point using a Breadth First Search (BFS) or a Depth First Search (DFS) algorithm. If we know where the endpoint is, the search is trivial. We can traverse backwards through the tree beginning at the endpoint back to the start point. Each path in an AST can represent an infinite number of code paths because a loop is represented by one node in the tree.

CGs or (I)CFGs are directed graphs in which cycles can occur. An infinite number of paths is possible in a graph with cycles, because cycles can be run infinitely often one after the other. Cycles are examples of Strongly Connected Components (SCC). If we replace cycles in the graph by one single node, we create a graph without cycles: a directed acyclic graph (DAG). The process of replacing a SCC by one single node is known as contracting. Paths in a DAG can be searched efficiently using a lexicographical search algorithm [44]. A path in a CG or (I)CFG can represent an infinite number of code paths. There is therefore a danger of state explosion.

7.4.2. SYMBOLIC EXECUTION

King [29] introduces the idea of Symbolic Execution. The path condition is a Boolean expression of symbolic inputs. The evaluation of path condition is considered feasible only if the symbolic variables are of the integer type and the expressions are linear combinations of the symbolic variables. Path conditions resulting from the guard of a while loop are not mentioned.

Burgstaller [10] presents a generic mathematical framework that can be used for imperative programming languages. The framework can define all variables bound to a given point in the program. The path expression algebra uses the supercontext to model the control flow information.

Li [33] et al show how SE can be used when searching for vulnerabilities in computer software. The location of a vulnerability (hotspot) can only be exploited if the hotspot can be reached (the program constraint PC is met) and the conditions to safely execute the code of the hotspot are not met (the security constraints SC are not met): $PC \wedge \neg SC$.

Dinges [18] et al combine in their research symbolic backward execution with forward concrete execution. They use this 'symcretic' method to efficiently find inputs that activate

¹<https://github.com/nikic/php-ast/blob/master/README.md#installation>

a specific branch statement of a Java program. Compared to concolic testing, their method provides input values that trigger the desired code components. The method is more efficient in the sense that fewer paths need to be explored. For solving path conditions the Z3 SMT solver is used.

Because path conditions have to describe all possible conditions that symbolic value variables have to meet, they can be too general. Smaragdakis [45] et al show that path conditions can be improved by combining these path conditions with concrete input values. Only input values that meet the path condition are used. The set of selected input values is therefore more limited than with fuzzing. This approach is called Dynamic Symbolic Execution. Loops, which are potentially run infinitely often, are abstracted by formulating pre and post conditions based on the conditions of the loop.

baldoni [5] et al provide an overview of techniques used in symbolic execution. In their analysis of research on SE, they give an overview of previous research that should remove difficulties of SE. They give possibilities to counter the problem of path explosion (pruning unreliable paths). Furthermore, they provide strategies that can further increase the use of SMT-solvers (among recent advances). As application areas of SE, they provide input generation, bug detection, bug exploitation and authentication bypass.

- CHEF, PHP interpreter anpassen
- concholic testing
- selective execution
- dynamic execution
- Abstract execution

7.4.3. TOOLS RESULTING FROM PREVIOUS RESEARCH

STRANGER is a tool resulting from research by Yu [56]. The analysis tool detects string-related vulnerabilities. A Control Flow Graph is constructed by using Pixy. A dependency graph is extracted from the CFG, by analysing the input flow with respect to string operations. Set of strings and string manipulations are modeled as deterministic finite automata. Cycles are replaced by one single strongly connected component. The tool is used to analyse a large scale PHP-project: SimpGB.

Symbolic pathfinder [41] is a tool that generates test cases from Java Bytecode by combining symbolic execution, model checking and constraint solving. From a conditional statement, path conditions are drawn up in a non-deterministic manner. These are evaluated with off-the-shelf constraint solvers.

SaferPHP [46] is a tool for discovering vulnerabilities in PHP-code. The tool can find DoS vulnerabilities and missing authorisation checks. It uses the PHP compiler to parse the code and to produce an AST. A Control flow graph (CFG) is made for each function. An interpreter in SaferPHP evaluates PHP-statements using symbolic values. Twenty binary operators (such as +, *, /,) are evaluated. Ten String-functions (such as substr, strlen) are evaluated symbolically as well.

Driller [48] is a tool that combines fuzzing and concolic testing. First, the fuzzer generates input values. These input values can be used to run the program through to the first complex checks. Now the program can try to generate input values that do meet these checks.

CHEF [8] tackles the problem of symbolic execution in a different way. The PHP interpreter is adapted in such a way that statement can also be executed symbolically. As a result, there is immediate support for all language constructions and there is no need to write a SE engine. The resulting tool is used for bug detection and exception exploration.

7.4.4. RESEARCH RELATED TO SMT SOLVERS

Vanegue [52] et al describe how SMT solvers can be used for vulnerability checking and automated exploit generation. SMT solvers can determine whether there is a solution to the path conditions. In that case the SMT solver can try to give a sample value. Modern SMT solvers can evaluate not only integer expressions but also expressions of string values. HAMPI [21] evaluates "membership over regular expressions, context-free grammar and the (in)equality of string values". Zheng [57] extends the SMT solver Z3 so that it can evaluate not only strings, but also expressions of integers and Boolean values. Statements about the length of strings can also be evaluated.

A

EXAMPLE OF PHP CODE VULNERABLE TO INJECTION

Example of PHP code that retrieves data from a database. This code is vulnerable to injection because the input from line 12 is used without sanitization in the query on line 14.

```
1  <?php
2  // connect to the database
3  $servername = "localhost";
4  $username = "root";
5  $password = "aPassword";
6  $database = "webshop";
7  mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);
8  $conn = new mysqli($servername, $username, $password, $database);
9  //echo "Connected successfully<br>";
10
11 // retrieve table
12 $productdesc = $_GET["productdesc"]
13 $query = "select * from product where productdesc='$productdesc'";
14 $queryResult = $conn->query($query);
15 echo "<table>";
16 while ($queryRow = $queryResult->fetch_row()) {
17     echo "<tr>";
18     for($i = 0; $i < $queryResult->field_count; $i++){
19         echo "<td>$queryRow[$i]</td>";
20     }
21     echo "</tr>";
22 }
23 echo "</table>";
24 $conn->close();
25 ?>
```

Listing A.1: example script for retrieving data from a database

B

EXAMPLE OF HOW A SAMATE PHP FILE TRANSLATES INTO SMTLIB

Line	PHP	SMTLIB	Remark
1		(set-option :produce-models true)	
2			
3	\$array = array();	(declare-const ia0 (Array Int String))	Declare array ia[Int]:String
4		(assert (= ia0 ((as const(Array Int String)) "")))	Fill ia with empty strings
5			
6	\$array[] = 'safe' ;	(declare-const s1 String)	declare s1:String
7		(assert (= s1 "safe"))	assign the value "safe" to s1
8		(declare-const ia1 (Array Int String))	declare a new array ia1[Int]:String
9		(assert (= ia1 (store ia0 0 s1)))	Put s1 at index 0 of ia0, store the result in ia1
10			
11	\$array[] = _GET['userData'] ;	(declare-const s2 String)	declare s2:String
12		(assert (= s2 "get"))	leave s2 empty
13		(declare-const ia2 (Array Int String))	declare a new array ia2[Int]:String
14		(assert (= ia2 (store ia1 1 s2)))	Put s2 at index 1 of ia1, store the result in ia2
15			
16	\$array[] = 'safe' ;	(declare-const s3 String)	declare s3:String
17		(assert (= s3 "safe"))	assign the value "safe" to s3
18		(declare-const ia3 (Array Int String))	declare a new array ia3[Int]:String
19		(assert (= ia3 (store ia2 2 s3)))	Put s3 at index 2 of ia2, store the result in ia3
20			
21	\$tainted = array[1] ;	(declare-const tainteds String)	declare tainteds:String
22		(assert (= (select ia3 1) tainteds))	Assign the value at index 1 of ia3 to tainteds
23			
24	\$tainted = (float)tainted ;	(declare-const taintedi Int)	declare taintedi:Int
25		(assert (= (str.to.int tainteds) taintedi))	cast the value of tainteds to in, store it in taintedi
26		(assert (>taintedi 0))	assert that taintedi >0
27			
28	echo \$tainted ;		echo
29			
30		(check-sat)	check whether the constraints are satisfiable
31		(get-model)	get a model

Table B.1: Example of a translation of PHP to SMTLIB

C

COMPARISON OF PHPAST AND PHPLY

PHP_AST	PHPLY
AST_ARRAY_ELEM: value, key	ArrayElement = node('ArrayElement', ['key', 'value', 'is_ref'])
AST_ARROW_FUNC: name, docComment, params, stmts, returnType, attributes	
AST_ASSIGN: var, expr	Assignment = node('Assignment', ['node', 'expr', 'is_ref'])
AST_ASSIGN_OP: var, expr	AssignOp = node('AssignOp', ['op', 'left', 'right'])
AST_ASSIGN_REF: var, expr	
AST_ATTRIBUTE: class, args // php 8.0+ attributes (version 80+)	ClassVariables = node('ClassVariables', ['modifiers', 'nodes'])
	ClassVariable = node('ClassVariable', ['name', 'initial'])
	ObjectProperty = node('ObjectProperty', ['node', 'name'])
	StaticProperty = node('StaticProperty', ['node', 'name'])
AST_BINARY_OP: left, right	BinaryOp = node('BinaryOp', ['op', 'left', 'right'])
AST_BREAK: depth	Break = node('Break', ['node'])
AST_CALL: expr, args	FunctionCall = node('FunctionCall', ['name', 'params'])
	MethodCall = node('MethodCall', ['node', 'name', 'params'])
AST_CAST: expr	Cast = node('Cast', ['type', 'expr'])
AST_CATCH: class, var, stmts	Catch = node('Catch', ['class_', 'var', 'nodes'])
AST_CLASS: name, docComment, extends, implements, stmts	Class = node('Class', ['name', 'type', 'extends', 'implements', 'traits', 'nodes'])
AST_CLASS_CONST: class, const	ClassConstant = node('ClassConstant', ['name', 'initial'])
AST_CLASS_CONST_GROUP: class, attributes // version 80+	ClassConstants = node('ClassConstants', ['nodes'])
AST_CLASS_NAME: class // version 70+	
AST_CLONE: expr	Clone = node('Clone', ['node'])
AST_CLOSURE: name, docComment, params, uses, stmts, returnType, attributes	Closure = node('Closure', ['params', 'vars', 'nodes', 'is_ref'])
AST_CLOSURE_VAR: name	
AST_CONDITIONAL: cond, true, false	Constant = node('Constant', ['name'])
AST_CONST: name	ConstantDeclaration = node('ConstantDeclaration', ['name', 'initial'])
AST_CONST_ELEM: name, value, docComment	
AST_CONTINUE: depth	Continue = node('Continue', ['node'])
AST_DECLARE: declares, stmts	Declare = node('Declare', ['directives', 'node'])
AST_DIM: expr, dim	
AST_DO_WHILE: stmts, cond	DoWhile = node('DoWhile', ['node', 'expr'])
AST_ECHO: expr	Echo = node('Echo', ['nodes'])
AST_EMPTY: expr	Empty = node('Empty', ['expr'])
AST_EXIT: expr	Exit = node('Exit', ['expr'])
AST_FOR: init, cond, loop, stmts	For = node('For', ['start', 'test', 'count', 'node'])
AST_FOREACH: expr, value, key, stmts	Foreach = node('Foreach', ['expr', 'keyvar', 'valvar', 'node'])
	ForeachVariable = node('ForeachVariable', ['name', 'is_ref'])
AST_FUNC_DECL: name, docComment, params, stmts, returnType, attributes	Function = node('Function', ['name', 'params', 'nodes', 'is_ref'])
	Method = node('Method', ['name', 'modifiers', 'params', 'nodes', 'is_ref'])
uses // prior to version 60	
AST_GLOBAL: var	Global = node('Global', ['nodes'])
AST_GOTO: label	
AST_GROUP_USE: prefix, uses	x
AST_HALT_COMPILER: offset	x
AST_IF_ELEM: cond, stmts	
	Elseif = node('Elseif', ['expr', 'node'])
	Else = node('Else', ['node'])
AST_INCLUDE_OR_EVAL: expr	Eval = node('Eval', ['expr'])
	Require = node('Require', ['expr', 'once'])
	Include = node('Include', ['expr', 'once'])
AST_INSTANCEOF: expr, class	
AST_ISSET: var	IsSet = node('IsSet', ['nodes'])
AST_LABEL: name	
AST_MAGIC_CONST:	MagicConstant = node('MagicConstant', ['name', 'value'])
AST_MATCH: cond, stmts // php 8.0+ match	
AST_MATCH_ARM: cond, expr // php 8.0+ match	
AST_METHOD: name, docComment, params, stmts, returnType, attributes	
uses // prior to version 60	
AST_METHOD_CALL: expr, method, args	
AST_METHOD_REFERENCE: class, method	
AST_NAME: name	
AST_NAMED_ARG: name, expr // php 8.0 named parameters	
AST_NAMESPACE: name, stmts	Namespace = node('Namespace', ['name', 'nodes'])
AST_NEW: class, args	New = node('New', ['name', 'params'])
AST_NULLABLE_TYPE: type // Used only since PHP 7.1	
AST_NULLSAFE_METHOD_CALL: expr, method, args // php 8.0 null safe operator	
AST_NULLSAFE_PROP: expr, prop // php 8.0 null safe operator	
AST_PARAM: type, name, default, attributes, docComment	Parameter = node('Parameter', ['node', 'is_ref'])
AST_POST_DEC: var	PreIncDecOp = node('PreIncDecOp', ['op', 'expr'])
AST_POST_INC: var	PostIncDecOp = node('PostIncDecOp', ['op', 'expr'])
AST_PRE_DEC: var	** zie hierboven
AST_PRE_INC: var	** zie hierboven
AST_PRINT: expr	Print = node('Print', ['node'])
AST_PROP: expr, prop	
AST_PROP_ELEM: name, default, docComment	
AST_PROP_GROUP: type, props, attributes // version 70+	
AST_REF: var // only used in foreach (\$a as &\$v)	
AST_RETURN: expr	Return = node('Return', ['node'])
AST_SHELL_EXEC: expr	
AST_STATIC: var, default	Static = node('Static', ['nodes'])
AST_STATIC_CALL: class, method, args	StaticMethodCall = node('StaticMethodCall', ['class_', 'name', 'params'])
AST_STATIC_PROP: class, prop	StaticVariable = node('StaticVariable', ['name', 'initial'])
AST_SWITCH: cond, stmts	Switch = node('Switch', ['expr', 'nodes'])
AST_SWITCH_CASE: cond, stmts	Case = node('Case', ['expr', 'nodes'])

Table C.1: List of AST nodes for PHPAST and PHPLY

PHP_AST	PHPLY	
AST_THROW: expr	Throw = node('Throw', ['node'])	
AST_TRAIT_ALIAS: method, alias	TraitUse = node('TraitUse', ['name', 'renames'])	
AST_TRAIT_PRECEDENCE: method, insteadof	TraitModifier = node('TraitModifier', ['from', 'to', 'visibility'])	
AST_TRY: try, catches, finally	Try = node('Try', ['nodes', 'catches', 'finally'])	
AST_TYPE:		
AST_UNARY_OP: expr	UnaryOp = node('UnaryOp', ['op', 'expr'])	
AST_UNPACK: expr		
AST_UNSET: var	Unset = node('Unset', ['nodes'])	
AST_USE_ELEM: name, alias		
AST_USE_TRAIT: traits, adaptations		
AST_VAR: name		
AST_WHILE: cond, stmts	While = node('While', ['expr', 'node'])	
AST_YIELD: value, key	Yield = node('Yield', ['node'])	
AST_YIELD_FROM: expr		
// List nodes (numerically indexed children):		
AST_ARG_LIST		
AST_ARRAY	Array = node('Array', ['nodes'])	
AST_ATTRIBUTE_LIST // php 8.0+ attributes (version 80+)		
AST_ATTRIBUTE_GROUP // php 8.0+ attributes (version 80+)		
AST_CATCH_LIST		
AST_CLASS_CONST_DECL		
AST_CLOSURE_USES		
AST_CONST_DECL		
AST_ENCAPS_LIST // interpolated string: "foo\$bar"		
AST_EXPR_LIST		
AST_IF	If = node('If', ['expr', 'node', 'elseifs', 'else_'])	
AST_LIST		
AST_MATCH_ARM_LIST // php 8.0+ match		
AST_NAME_LIST		
AST_PARAM_LIST		
AST_PROP_DECL		
AST_STMT_LIST	Block = node('Block', ['nodes'])	
AST_SWITCH_LIST		
AST_TRAIT_ADAPTATIONS		
AST_USE		
AST_TYPE_UNION // php 8.0+ union types		
	Silence = node('Silence', ['expr'])	
	Finally = node('Finally', ['nodes'])	
	Default = node('Default', ['nodes'])	
	Trait = node('Trait', ['name', 'traits', 'nodes'])	
	InlineHTML = node('InlineHTML', ['data'])	
	UseDeclarations = node('UseDeclarations', ['nodes'])	
	UseDeclaration = node('UseDeclaration', ['name', 'alias'])	
	ListAssignment = node('ListAssignment', ['nodes', 'expr'])	
	TernaryOp = node('TernaryOp', ['expr', 'iftrue', 'iffalse'])	
	ConstantDeclarations = node('ConstantDeclarations', ['nodes'])	
	Directive = node('Directive', ['name', 'node'])	
	Variable = node('Variable', ['name'])	
	LexicalVariable = node('LexicalVariable', ['name', 'is_ref'])	
	MethodCall = node('MethodCall', ['node', 'name', 'params'])	
	ArrayOffset = node('ArrayOffset', ['node', 'expr'])	
	StringOffset = node('StringOffset', ['node', 'expr'])	

Table C.2: List of AST nodes for PHPAST and PHPLY

D

OPERATORS FOR TYPES IN SMTLIB

	Bool	Int	Real	BitVec	String
<i>arithmetic</i>					
addition		+	+	bvadd	
subtraction		-	-	bvsub	
division		/	/		
division		div		bvdiv	
multiplication		*	*	bvmul	
unsigned div		div		bvudiv	
unsigned rem		mod		bvurem	
shl				bvshl	
unsigned logical sht				bvlshr	
Signed arithmetic shr				bvashr	
unary negation		-	-	bvneg	

<i>bitwise operators</i>					
bitwise and	and			bvand	
bitwise or	or			bvor	
implication					
bitwise not	=>			bvnot	
bitwise nand				bvnand	
bitwise nor				bvnor	
bitwise xnor				bvxnor	

<i>numerical comparison</i>					
=		=	=		
Signed <		<	<	bvult	
Signed <=		<=	<=	bvule	
Signed >		>	>	bvugt	
Signed >=		>=	>=	bvuge	
Unsigned <				bvslt	
Unsigned <=				bvsle	
Unsigned >				bvsgt	
Unsigned >=				bvsge	

<i>lexicographical order</i>					
<					Str.<
=					
<=					<=

	Bool	Int	Real	BitVec	String
<i>string functions</i>					
concat					str.++
length					str.len
at					str.at
substr					str.substr
prefixof					str.prefixof
suffixof					str.suffixof
contains					str.contains
indexof					str.indexof
replace					str.replace
replace_all					str.replace_all
is_digit					str.is_digit

<i>conversion/cast</i>					
from_int					from_int
to_int					cast to int
to.int					str_to_int
int.to.str					int_to_str
cast to float					to_real

E

UTILITIES

In addition to the main tool, we have developed a number of utilities to support the use of the main tool.

E.1. SAMATE LINE NUMBER INSERTER

The SAMATE test set provides the line number containing the vulnerability of unsafe samples in a separate XML file. This tool adds the line number to the comment section at the beginning of the PHP file, so that the line number containing the vulnerability is immediately available when reading the PHP file.

E.2. SAMATE CWE SELECTOR

The beginning of a file name in the SAMATE collection indicates for which CWE vulnerability it contains a sample. The utility uses this part of the filename to select all files belonging to a certain CWE number and collects them in a folder.

In addition, the source file indicates whether it is a safe or unsafe sample. The utility can use this line in the source file to break down the files for a CWE by sample type (safe/unsafe) if required.

```
1 <?php
2 /*
3 Unsafe sample
4 input : use fopen to read /tmp/tainted.txt and put the first ...
5 sanitize : use of the function htmlentities. Sanitizes the qu ...
6 construction : concatenation with simple quote
7 */
```

Listing E.1: example of a PHP source file from the SAMATE dataset, showing the sample type (safe/unsafe) in the comment section

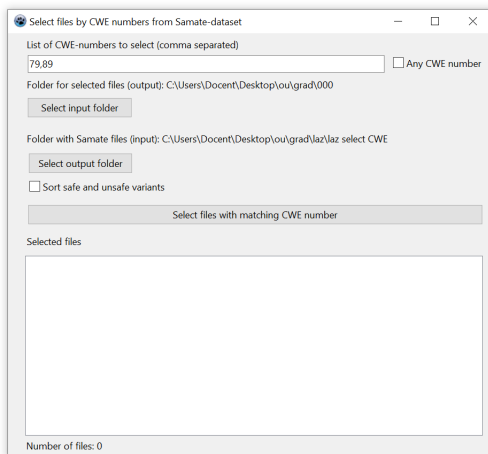


Figure E.1: Utility for preparing a subset of the SAMATE dataset

E.3. REMOVE SPECIAL CHARACTERS FROM FILE NAME

When creating an AST file from a PHP file, the percentage symbol is removed from the file name. Since the main tool requires that the PHP file and the corresponding AST file have the same name (apart from the extension), we had to use this utility to remove special characters (such as the percentage symbol) from the file name so that this condition is met.¹

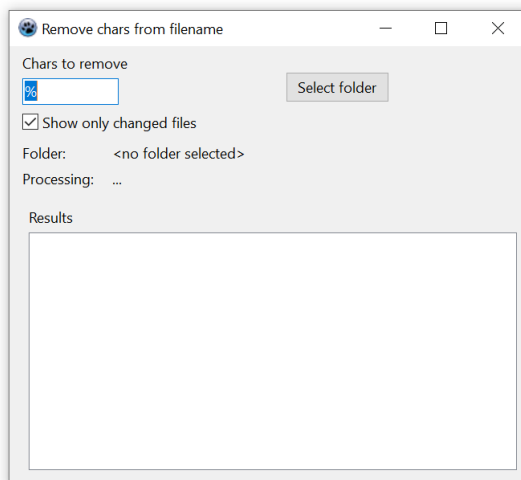


Figure E.2: Utility for removing special characters from filenames

¹<https://github.com/rdohmen/remove-special-chars>

E.4. BATCH FILE CREATOR

The main tool needs an AST file for each PHP file. We create these AST files by passing the file name of the PHP source code as an argument to PHP script called `astarg.php`. This is done for one file at a time. This utility can create a batch file that calls `astarg.php` for a series of PHP files. In fact, the tool can be used for any situation where a series of files needs to be passed as argument to a command line tool.²

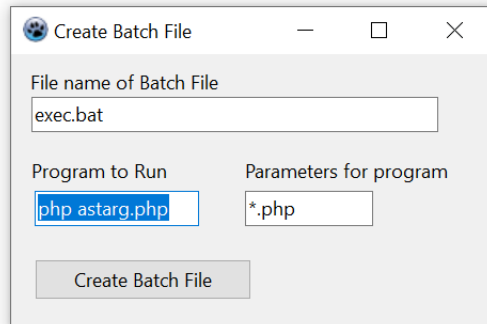


Figure E.3: Utility for creating a batch file that produces all AST files

E.5. TESTING TOOL FOR GRAPH UNIT

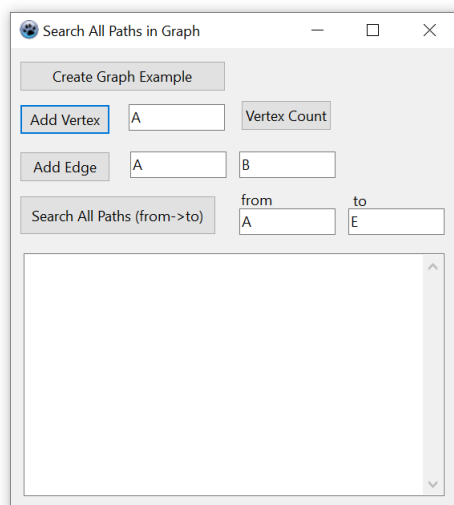


Figure E.4: Unit test for the Graph unit

²<https://github.com/rdohmen/batch-file-creator>

E.6. TOOL FOR EXTRACTING AST FUNCTIONS

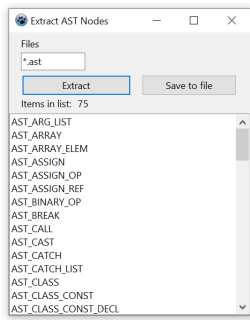


Figure E.5: Tool for extracting AST nodes from AST files

E.7. TESTING TOOL FOR TYPE INFERENCE UNIT

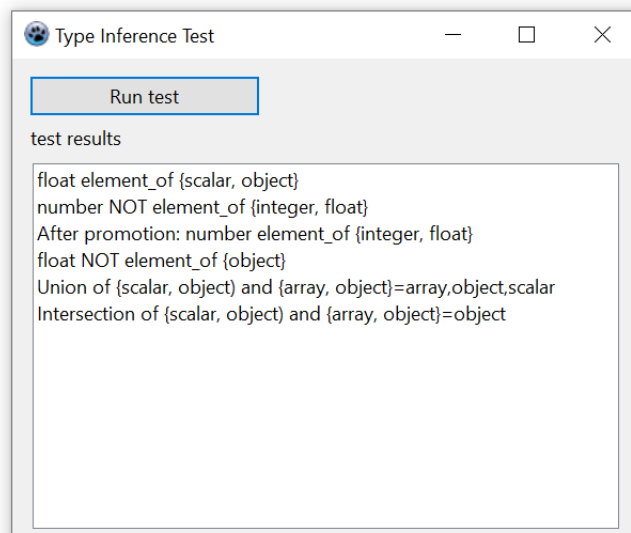


Figure E.6: Unit test for the type inference unit

E.8. TESTING TOOL FOR TESTING AST FUNCTIONS

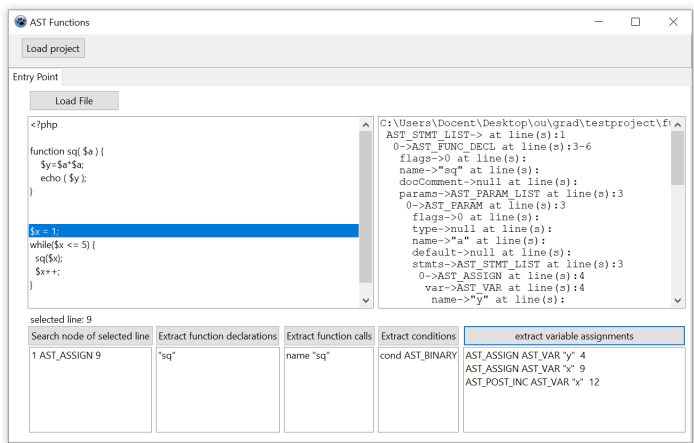


Figure E.7: Tool for testing various AST functions

BIBLIOGRAPHY

- [1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. “Compilers, principles, techniques”. In: *Addison wesley* 7.8 (1986), p. 9.
- [2] Mahtab Alam. “Software security requirements checklist”. In: *International Journal of Software Engineering, IJSE* 3.1 (2010), pp. 53–62.
- [3] Andrew W Appel. “SSA is functional programming”. In: *Acm Sigplan Notices* 33.4 (1998), pp. 17–20.
- [4] John Aycock and Nigel Horspool. “Simple generation of static single-assignment form”. In: *International Conference on Compiler Construction*. Springer. 2000, pp. 110–125.
- [5] Roberto Baldoni et al. “A survey of symbolic execution techniques”. In: *ACM Computing Surveys (CSUR)* 51.3 (2018), pp. 1–39.
- [6] R.M.K Beisicht. “Injection Attack Mitigation; A secure multi-execution approach”. MA thesis. Open University, the Netherlands, 2019.
- [7] Matthias Braun et al. “Simple and efficient construction of static single assignment form”. In: *International Conference on Compiler Construction*. Springer. 2013, pp. 102–122.
- [8] Stefan Bucur, Johannes Kinder, and George Candea. “Prototyping symbolic execution engines for interpreted languages”. In: *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 2014, pp. 239–254.
- [9] Bernd Burgstaller, Bernhard Scholz, and Johann Blieberger. “A symbolic analysis framework for static analysis of imperative programming languages”. In: *Journal of systems and software* 85.6 (2012), pp. 1418–1439.
- [10] Bernd Burgstaller, Bernhard Scholz, and Johann Blieberger. “Symbolic analysis of imperative programming languages”. In: *Joint Modular Languages Conference*. Springer. 2006, pp. 172–194.
- [11] Vitaly Chipounov et al. “Selective symbolic execution”. In: *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*. CONF. 2009.
- [12] Stephen A Cook. “The complexity of theorem-proving procedures”. In: *Proceedings of the third annual ACM symposium on Theory of computing*. 1971, pp. 151–158.
- [13] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. “A simple, fast dominance algorithm”. In: *Software Practice & Experience* 4.1-10 (2001), pp. 1–8.
- [14] Martin Davis, George Logemann, and Donald Loveland. “A machine program for theorem-proving”. In: *Communications of the ACM* 5.7 (1962), pp. 394–397.
- [15] Martin Davis and Hilary Putnam. “A computing procedure for quantification theory”. In: *Journal of the ACM (JACM)* 7.3 (1960), pp. 201–215.

- [16] Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. “A practical interprocedural dominance algorithm”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29.4 (2007), 19–es.
- [17] Dominique Devriese and Frank Piessens. “Noninterference through secure multi-execution”. In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 109–124.
- [18] Peter Dinges and Gul Agha. “Targeted test input generation using symbolic-concrete backward execution”. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 31–36.
- [19] B.W.A. Elema. “Finding Chinks in the armour; Software vulnerability prediction using deep learning on graph representation of source code”. MA thesis. Open University, the Netherlands, 2020.
- [20] Daniele Filaretti and Sergio Maffei. “An executable formal semantics of PHP”. In: *European Conference on Object-Oriented Programming*. Springer. 2014, pp. 567–592.
- [21] Vijay Ganesh et al. “HAMPI: A string solver for testing, analysis and vulnerability detection”. In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 1–19.
- [22] Patrice Godefroid. “Fuzzing: Hack, art, and science”. In: *Communications of the ACM* 63.2 (2020), pp. 70–76.
- [23] David Grove and Craig Chambers. *An assessment of call graph construction algorithms*. IBM Thomas J. Watson Research Division, 2000.
- [24] C. L. Hamblin. “Translation to and from Polish Notation”. In: *The Computer Journal* 5.3 (Nov. 1962), pp. 210–213. ISSN: 0010-4620. DOI: [10.1093/comjnl/5.3.210](https://doi.org/10.1093/comjnl/5.3.210). eprint: <https://academic.oup.com/comjnl/article-pdf/5/3/210/1172943/5-3-210.pdf>. URL: <https://doi.org/10.1093/comjnl/5.3.210>.
- [25] Mark Hills, Paul Klint, and Jurgen J Vinju. “Enabling PHP software engineering research in Rascal”. In: *Science of Computer Programming* 134 (2017), pp. 37–46.
- [26] Roger Hindley. “The principal type-scheme of an object in combinatory logic”. In: *Transactions of the american mathematical society* 146 (1969), pp. 29–60.
- [27] Joxan Jaffar, Jorge A Navas, and Andrew E Santosa. “Unbounded symbolic execution for program verification”. In: *International Conference on Runtime Verification*. Springer. 2011, pp. 396–411.
- [28] Gary A Kildall. “A unified approach to global program optimization”. In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1973, pp. 194–206.
- [29] James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: [10.1145/360248.360252](https://doi.org/10.1145/360248.360252). URL: <https://doi.org/10.1145/360248.360252>.
- [30] Samuel Klingström. “Type Inference in PHP using Deep Learning”. MA thesis. Lunds University, Sweden, 2020.
- [31] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2016.

- [32] J.J. Kronjee. “DISCOVERING VULNERABILITIES USING DATA-FLOW ANALYSIS AND MACHINE LEARNING DEMONSTRATED FOR PHP APPLICATIONS”. MA thesis. Open University, the Netherlands, 2020.
- [33] Hongzhe Li et al. “Software vulnerability detection using backward trace analysis and symbolic execution”. In: *2013 International Conference on Availability, Reliability and Security*. IEEE. 2013, pp. 446–454.
- [34] Ken Madlener, Sjaak Smetsers, and Marko Van Eekelen. “A Formal Verification Study on the Rotterdam Storm Surge Barrier”. In: *Proceedings of the 12th International Conference on Formal Engineering Methods and Software Engineering*. ICFEM’10. Shanghai, China: Springer-Verlag, 2010, pp. 287–302. ISBN: 3642169007.
- [35] Abdalla Wasef Marashdih, Zarul Fitri Zaaba, and Saman M Almufti. “The Problems and Challenges of Infeasible Paths in Static Analysis”. In: *International Journal of Engineering & Technology* 7.4.19 (2018), pp. 412–417.
- [36] Thomas J McCabe. “A complexity measure”. In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.
- [37] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.
- [38] Glenford J Myers. *The art of software testing*. John Wiley & Sons, 2006.
- [39] Toyoshiro Nakashima et al. “Analysis of software bug causes and its prevention”. In: *Information and Software technology* 41.15 (1999), pp. 1059–1068.
- [40] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. “Abstract DPLL and abstract DPLL modulo theories”. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 2005, pp. 36–50.
- [41] Corina S Păsăreanu and Neha Rungta. “Symbolic PathFinder: symbolic execution of Java bytecode”. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 2010, pp. 179–180.
- [42] Frank Piessens. “A taxonomy of causes of software vulnerabilities in internet software”. In: *Supplementary Proceedings of the 13th International Symposium on Software Reliability Engineering*. Citeseer. 2002, pp. 47–52.
- [43] John Alan Robinson. “A machine-oriented logic based on the resolution principle”. In: *Journal of the ACM (JACM)* 12.1 (1965), pp. 23–41.
- [44] Steven S. Skiena. *The Algorithm Design Manual*. Springer, 2009.
- [45] C Csallner–N Tillmann–Y Smaragdakis. “DySy: Dynamic Symbolic Execution for Invariant Inference”. In: (2008).
- [46] Sooel Son and Vitaly Shmatikov. “SAFERPHP: Finding semantic vulnerabilities in PHP applications”. In: *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*. 2011, pp. 1–13.
- [47] *SQLinjection*. https://www.researchgate.net/figure/Example-of-a-SQL-Injection-Attack_fig1_265947554. Accessed: 2021-03-01.
- [48] Nick Stephens et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution.” In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

- [49] Bertrand Stivalet and Elizabeth Fong. “Large scale generation of complex and faulty PHP test cases”. In: *2016 IEEE International conference on software testing, verification and validation (ICST)*. IEEE. 2016, pp. 409–415.
- [50] Peri L Tarr and Alexander L Wolf. *Engineering of Software: The Continuing Contributions of Leon J. Osterweil*. Springer Science & Business Media, 2011.
- [51] M.N.S. Thulasiraman K; Wamy. *Graphs: Theory and algorithms*. John Wiley and Son, 1992.
- [52] Julien Vanegue, Sean Heelan, and Rolf Rolles. “SMT Solvers in Software Security.” In: *WOOT 12 (2012)*, pp. 9–22.
- [53] Ruud Weijde van. “Type inference for PHP; A constraint based type inference written in Rascal”. MA thesis. Open University, the Netherlands, 2020.
- [54] XSS cross site scripting. <https://blogs.sap.com/2015/12/17/xss-cross-site-scripting-overview-with-contexts/>. Accessed: 2021-03-01.
- [55] Javad Yousefi, Yasser Sedaghat, and Mohammadreza Rezaee. “Masking wrong-successor Control Flow Errors employing data redundancy”. In: *2015 5th International Conference on Computer and Knowledge Engineering (ICCCKE)*. IEEE. 2015, pp. 201–205.
- [56] Fang Yu, Muath Alkhalaf, and Tefvik Bultan. “Stranger: An automata-based string analysis tool for php”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2010, pp. 154–157.
- [57] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. “Z3-str: A z3-based string solver for web application analysis”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 2013, pp. 114–124.

ARTICLES

- [1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. “Compilers, principles, techniques”. In: *Addison wesley* 7.8 (1986), p. 9.
- [2] Mahtab Alam. “Software security requirements checklist”. In: *International Journal of Software Engineering, IJSE* 3.1 (2010), pp. 53–62.
- [3] Andrew W Appel. “SSA is functional programming”. In: *Acm Sigplan Notices* 33.4 (1998), pp. 17–20.
- [5] Roberto Baldoni et al. “A survey of symbolic execution techniques”. In: *ACM Computing Surveys (CSUR)* 51.3 (2018), pp. 1–39.
- [9] Bernd Burgstaller, Bernhard Scholz, and Johann Blieberger. “A symbolic analysis framework for static analysis of imperative programming languages”. In: *Journal of systems and software* 85.6 (2012), pp. 1418–1439.
- [13] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. “A simple, fast dominance algorithm”. In: *Software Practice & Experience* 4.1-10 (2001), pp. 1–8.
- [14] Martin Davis, George Logemann, and Donald Loveland. “A machine program for theorem-proving”. In: *Communications of the ACM* 5.7 (1962), pp. 394–397.
- [15] Martin Davis and Hilary Putnam. “A computing procedure for quantification theory”. In: *Journal of the ACM (JACM)* 7.3 (1960), pp. 201–215.
- [16] Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. “A practical interprocedural dominance algorithm”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29.4 (2007), 19–es.
- [22] Patrice Godefroid. “Fuzzing: Hack, art, and science”. In: *Communications of the ACM* 63.2 (2020), pp. 70–76.
- [24] C. L. Hamblin. “Translation to and from Polish Notation”. In: *The Computer Journal* 5.3 (Nov. 1962), pp. 210–213. ISSN: 0010-4620. DOI: [10.1093/comjnl/5.3.210](https://doi.org/10.1093/comjnl/5.3.210). eprint: <https://academic.oup.com/comjnl/article-pdf/5/3/210/1172943/5-3-210.pdf>. URL: <https://doi.org/10.1093/comjnl/5.3.210>.
- [25] Mark Hills, Paul Klint, and Jurgen J Vinju. “Enabling PHP software engineering research in Rascal”. In: *Science of Computer Programming* 134 (2017), pp. 37–46.
- [26] Roger Hindley. “The principal type-scheme of an object in combinatory logic”. In: *Transactions of the american mathematical society* 146 (1969), pp. 29–60.
- [29] James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: [10.1145/360248.360252](https://doi.org/10.1145/360248.360252). URL: <https://doi.org/10.1145/360248.360252>.
- [35] Abdalla Wasef Marashdih, Zarul Fitri Zaaba, and Saman M Almufti. “The Problems and Challenges of Infeasible Paths in Static Analysis”. In: *International Journal of Engineering & Technology* 7.4.19 (2018), pp. 412–417.

- [36] Thomas J McCabe. “A complexity measure”. In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.
- [37] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.
- [39] Toyoshiro Nakashima et al. “Analysis of software bug causes and its prevention”. In: *Information and Software technology* 41.15 (1999), pp. 1059–1068.
- [43] John Alan Robinson. “A machine-oriented logic based on the resolution principle”. In: *Journal of the ACM (JACM)* 12.1 (1965), pp. 23–41.
- [45] C Csallner–N Tillmann–Y Smaragdakis. “DySy: Dynamic Symbolic Execution for Invariant Inference”. In: (2008).
- [52] Julien Vanegue, Sean Heelan, and Rolf Rolles. “SMT Solvers in Software Security.” In: *WOOT* 12 (2012), pp. 9–22.

BOOKS

- [23] David Grove and Craig Chambers. *An assessment of call graph construction algorithms*. IBM Thomas J. Watson Research Division, 2000.
- [31] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2016.
- [38] Glenford J Myers. *The art of software testing*. John Wiley & Sons, 2006.
- [44] Steven S. Skiena. *The Algorithm Design Manual*. Springer, 2009.
- [50] Peri L Tarr and Alexander L Wolf. *Engineering of Software: The Continuing Contributions of Leon J. Osterweil*. Springer Science & Business Media, 2011.
- [51] M.N.S. Thulasiraman K; Wamy. *Graphs: Theory and algorithms*. John Wiley and Son, 1992.