

## Article

# An Approach for Detecting Feasible Paths Based on Minimal SSA Representation and Symbolic Execution

Abdalla Wasef Marashdih <sup>1</sup>, Zarul Fitri Zaaba <sup>1,\*</sup> and Khaled Suwais <sup>2</sup>

<sup>1</sup> School of Computer Sciences, Universiti Sains Malaysia, Pulau Pinang 11800, Malaysia; a.w.marashdeh@student.usm.my

<sup>2</sup> Faculty of Computer Studies, Arab Open University, P.O.Box 84901, Riyadh 11681, Saudi Arabia; khaled.suwais@arabou.edu.sa

\* Correspondence: zarulfitri@usm.my

**Abstract:** Static analysis is one of the techniques used today to analyze source codes and minimize the issue of software vulnerability. Static analysis has the ability to observe all possible software paths in an application through the scrutiny of a web application's source code. Among those paths, some may be considered feasible paths, which refer to any paths that the test cases can execute. The detection of feasible paths in the results of a static analysis helps to minimize the false positive rate. However, the detection of feasible paths can be challenging, especially for programs that have multiple conditions in the same branch. The aim is to ensure that each feasible path is detected only once (not duplicated). This paper proposes an approach based on minimal static single assignment (MSSA) form and symbolic execution to detect feasible paths. The proposed approach starts by converting the source code into an abstract syntax tree (AST), followed by converting the AST to minimal SSA representation, which helps to decrease the number of instructions in the SSA form. An algorithm was built to examine all of the instructions of the SSA form, identify whole paths in the source code, and extract constraints along each path. A path weight method (PWM) is proposed in this work to avoid detecting duplicated feasible paths. The satisfiability modulo theory (SMT) solver was used to check the satisfiability of each path condition. The proposed approach was tested on seven well-known test programs that have been used in related studies and 10 large scale programs. The experimental results indicate that the proposed method (PWM) can avoid detecting duplicated feasible paths, and the proposed approach reduced the time required for generating the paths compared to that in related studies.

**Keywords:** detection; feasible paths; SSA; static analysis; symbolic execution



**Citation:** Marashdih, A.W.; Zaaba, Z.F.; Suwais, K. An Approach for Detecting Feasible Paths Based on Minimal SSA Representation and Symbolic Execution. *Appl. Sci.* **2021**, *11*, 5384. <https://doi.org/10.3390/app11125384>

Academic Editors: Pietro Ferrara and Agostino Cortesi

Received: 15 April 2021

Accepted: 31 May 2021

Published: 10 June 2021

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Web applications are now considered one of the standard channels in the World Wide Web for representing data and executing service launches. These resources include social media, finance, education, banking, news pages, TV channels, etc. Web applications store sensitive data, which hackers can steal and gain monetary benefits from.

Thus, to identify the vulnerability of web applications, a static analysis explores the web application's source code [1]. Static analysis has the potential to observe every possible path in an application through scrutiny of the source code. However, covering all web application paths during testing leads a false positive result [2,3]. False positives are results that are safe but are reported to be vulnerabilities, and thus, execution would not occur, despite the type of user input. In addition, researchers have stated that programmers should distinguish which paths can be executed and that the test cases should be implemented on feasible paths only [4,5].

A feasible path is defined as any path on which test cases can be executed. Detection of feasible paths can help decrease the false positive rate of the results of static analyses and, thus, further improve the detection rate for different threats. In addition, detecting

the same feasible paths (duplicated paths) from test cases more than once provides a false number of threats existing in the program.

Detection of feasible paths can be achieved by generating test cases of the source code to identify the executable paths among the others. Symbolic execution is a basic approach commonly used in various fields, such as software testing and reverse engineering [6–8]. The idea is to execute a program with a symbolic representation of inputs instead of concrete values. The symbol signifies symbolic and concrete values that may be understood from the symbol. Some symbolic execution engines have been deployed. Prominent examples are MultiSE [6], KLEE [7], and Java PathFinder (Jpf) [8]. However, all of these studies focused on the intermediate representation (IR) to convert the source code to a simple form for analysis, such as the static single assignment (SSA) form. Nevertheless, there is significant inflation in the IR code, where new constructions correspond to function abstractions. For instance, the code written for the Low-Level Virtual Machine (LLVM) occupies 600 lines, despite the simplicity of the example [9,10]. The analysis of this code was time-consuming, as would be the case for similar examples. Meanwhile, Braun et al. [11] presented an SSA constriction algorithm to produce a minimal and pruned SSA form. They established that the algorithm they designed builds minimal and trimmed SSA form, which helps decrease the instructions in the SSA form to be analyzed.

Therefore, this paper aims to reinforce the improvements in the static analysis results by proposing a new approach for detecting feasible paths based on minimal SSA representation and symbolic execution. The proposed approach was constructed in such a way that it can (i) detect the total quantity of feasible paths in the source code, (ii) avoid detecting duplicated feasible paths, and (iii) decrease the time required to generate the paths. In addition, IR and binary generation tools and libraries are not commonly accessible for all programming languages, such as Hypertext Preprocessor (PHP), thereby significantly reducing the applicability of current symbolic execution systems [12]. Therefore, our approach was implemented and tested on PHP as the most popular web application technology [13].

The rest of the paper is structured as follows: Section 2 provides background information on static analysis and symbolic execution, followed by related studies in Section 3. Section 4 elucidates the proposed detection approach of feasible paths. Section 5 presents the experimental outcomes of the proposed approach and comparisons with related work. Section 6 highlights the threats to validity. Section 7 presents the discussion, and Section 8 concludes this work and introduces our upcoming work.

## 2. Background

### 2.1. Static Analysis

Static code analysis is one of the most widely used tools of source code analysis because it can identify potential security issues or weaknesses without program execution [14]. During static analysis, the complete source code is analyzed, which makes this tool very powerful in the detection of security-specific issues [15]. Nevertheless, the technique is not completely free of errors; false positives may still occur because the program traversed an infeasible path or another that failed execution. For these reasons, the concern regarding false positives is significant in the static code analysis context and requires additional evaluation. The removal of infeasible paths from the set of paths on which static analyses are performed is another potential solution to address the false positive concern [16].

Infeasible paths mean paths that will not be executed regardless of the input. Jiang et al. [17] mentioned that infeasible paths can waste time and resources in software testing. Meanwhile, a feasible path can be defined as any path in the source code that test cases can execute [18]. Figure 1 depicts a control flow graph (CFG) specific to a PHP source code example, showing a scenario in which feasible paths, duplicated paths and infeasible paths all exist in the test code.

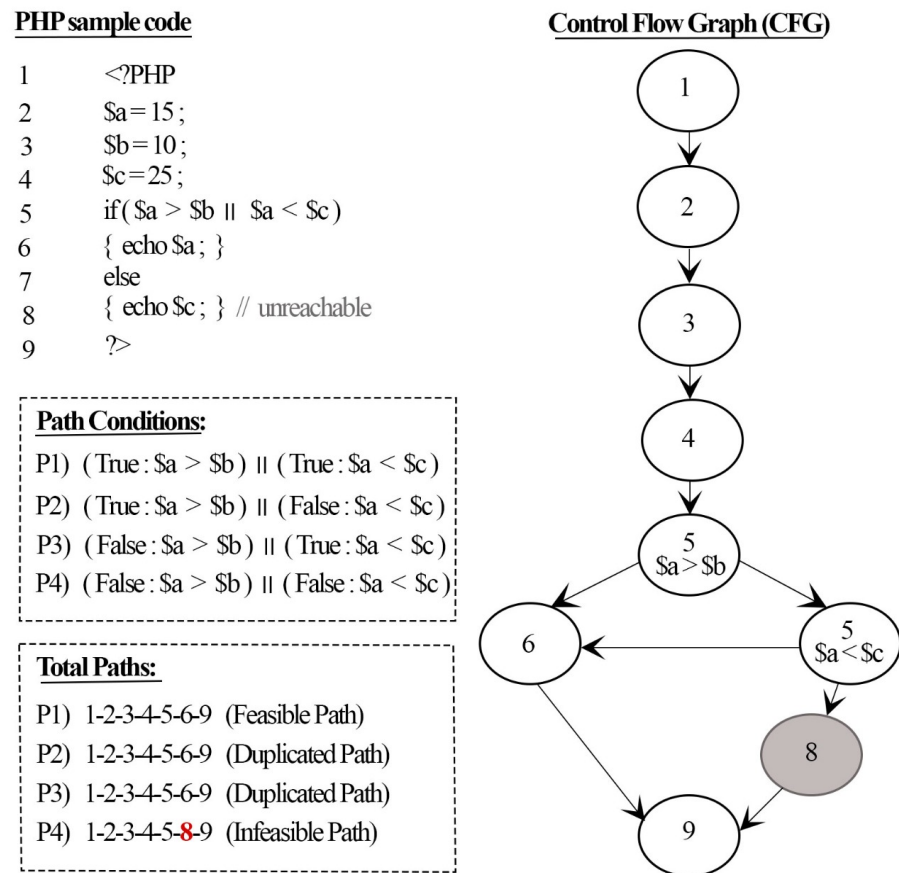


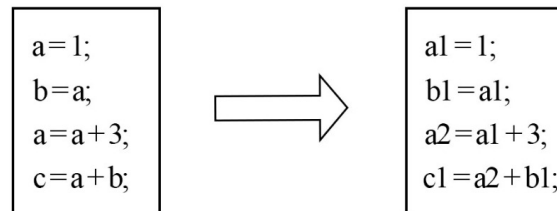
Figure 1. Example of infeasible path and duplicated paths in PHP source code.

Static analysis begins with source code analysis and a control flow graph that corresponds to the source code created. The source code for the above-mentioned PHP example begins with the initialization of variables on lines 2, 3 and 4 ( $a = 15$ ,  $b = 10$ , and  $c = 25$ ). Subsequently, two conditions on the same branch specified on line 5 determine whether any of the conditions ( $a > b$  or  $b < c$ ) are true. The conversion of this branch (line 5) in the CFG is split into separate branches depending on the number of conditions in that branch (our example was split into two separate branches). In the first path (P1), since both conditions are satisfied ( $a > b \ || \ b < c$ ), the code specified in line 6 (echo  $a$ ) is executed, and this path is considered a feasible path. In P2 and P3, the logical operator ( $\ || \$  ‘or’) will be true if one of the conditions is true. Therefore, P2 and P3 are also satisfied since one of their conditions is true based on the variable input ( $a > b$ ) or ( $a < c$ ). However, the paths (P1, P2, and P3) lead to the same path (1-2-3-4-5-6-9) and are considered duplicated paths. In the last path (P4), the flow of the program cannot be transferred to the “else” statement (line 7) because, in consideration of the values of variables  $a$ ,  $b$ , and  $c$ , the conditions are always “True”. The print statement (line 8) is unattainable under the given inputs. As shown in Figure 1, any path that contains the implementation of the “else” statement (line 8 in red color) is considered an infeasible path because it cannot be executed at all.

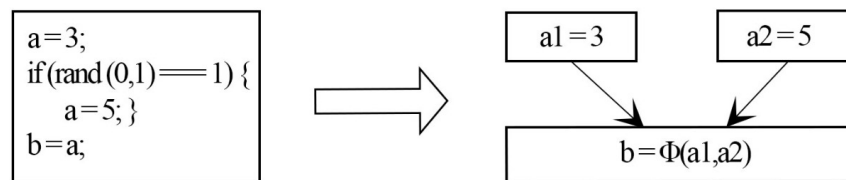
Infeasible paths could exist for several reasons [18]. One of these causes is dead code, which implies that the execution of a specific line of code is impossible. Infeasible paths are also caused by the conflicting clauses found within certain paths. Moreover, infeasible paths could potentially be associated with conditional statements specific to a variable (which is also one of the major causes of the inability to access specific aspects of a program [19,20]). Therefore, to improve the outcomes of the static analysis, a detection stage of the feasible paths within the source code should be in place.

The static single assignment (SSA) form [21] is source code expressed in the intermediate representation form under the condition that all variables are defined before use

and assigned only once, which can be achieved by renaming variables such that each variable name occurs only once on the left-hand side of an assignment statement. Figure 2a illustrates such variable renaming. However, this representation introduces a complication at join points of different flow branches. To solve this problem, a  $\phi$  function is used in places where we could not pinpoint the exact variable “version” [19,20]. A code corresponding to the  $\phi$  function usage and a simplified graph excerpt are shown in Figure 2b.



(a) Code transformation to SSA



(b) Code requiring a  $\Phi$  function

**Figure 2.** Example of static single assignment (SSA) representation and  $\phi$  function. (a) Variable renaming. (b) A code corresponding to the  $\phi$  function usage and a simplified graph excerpt.

As each variable is assigned once in the SSA representation and a  $\phi$  function is added at each join point, it leads to an increase in the time required to analyze all instructions of the program. For instance, the LLVM code for the simple example comprises over 600 lines [9], which requires considerable time to analyze the source code for such examples. Therefore, removing some of the instructions that will not affect the program execution and decreasing the number of the  $\phi$  functions in the SSA form would be solutions that would produce a minimal SSA form that can decrease the time taken to detect the feasible paths.

## 2.2. Symbolic Execution

Symbolic execution is the execution of a program with a symbolic representation of inputs instead of concrete values [22]. Concrete executors, for example, an interpreter, can only evaluate expressions consisting of concrete values. Meanwhile, symbolic executors allow the existence of symbolic values in the expressions, that is, expressions of concrete values as well as names that have not been given values. By allowing inputs supplied as symbolic values, a symbolic executor can construct a relation between inputs and outputs along any execution path.

To understand the general concept of symbolic execution, refer to Figure 3. If the symbolic value “A” is inputted to parameter “a” while the function named “Test” is under execution, it may be said that the function is being symbolically executed. When the code is being executed, the two “If” statements do not have control over program flow because “A” may assume different values depending on the context. The trick is to construct a logical expression that captures the relation between the \$res value and the input.

1	Test (\$a) {	P1: $A < 0 \wedge A > 10$	P1: (UNSAT)
2	\$res=0;	P2: $A < 0 \wedge A \leq 10$	P2: (SAT: $A=-1$ )
3	if (\$a<0) { \$res=-1; }	P3: $A \geq 0 \wedge A > 10$	P3: (SAT: $A=11$ )
4	if (\$a>10) { \$res=1; }	P4: $A \geq 0 \wedge A \leq 10$	P4: (SAT: $A=10$ )
5	return \$res;		
6	}		
	(a) PHP source code example	(b) Paths conditions	(c) Constraint Solver

**Figure 3.** Path conditions example in symbolic execution.

This is a powerful program description that discloses all of the important factors in an input to the PHP example program as shown in Figure 3a. It can be understood as a specification of path constraint and the exhibited behavior for each feasible program path. A path constraint indicates the condition for input, in this case,  $A$ , to meet so that the execution will go down a particular path. For example, in Figure 3b, the path goes through the false branches of both if statements have a path constraint of  $(A \geq 0 \wedge A \leq 10)$ , and the exhibited behavior is given by  $\$res = 0$ . The path constraint has an assignment ( $\$res = 0, X = 0$ ) under which the constraint is evaluated as “True”, which makes this path feasible. Meanwhile, the path constraint for the path goes through the true branches of both if statements are infeasible because it requires  $(X < 0 \wedge X > 10)$ , which is unsatisfiable.

A typical symbolic execution handles each path separately in a symbolic state [22]. It is essentially a container of a path constraint and records the program behaviors that symbolic execution propagates, such as the state towards the end of each path, during which symbolic execution identifies feasible and infeasible paths by asking a constraint solver. Constraint solvers are employed to determine the reason specific to symbolic expressions automatically. Constraint solving is a technique that determines a set of values for the variables that satisfy an expression subject to several constraints concerning the variables. In our example in Figure 3c, four different paths can be observed, three of which are feasible. Symbolic execution finds a conflict in the first path constraint (P1), where the constraint solver returns unsatisfiable for this path because it does not represent a program behavior that can happen. If constraint solving returns a satisfying value assignment for a given constraint instead, then the path is feasible, and we can use the found values as test inputs to re-exhibit the associated program behavior.

### 3. Related Works

In recent years, various approaches have been utilized to detect feasible paths. We point out some works in this section that are pertinent to our study. Directed automated random testing (DART) [23] was the first concolic testing study used to reduce the number of test cases to generate program paths. The paths are generated randomly based on the type of the variables, and the conditions of each path are stored until the termination of the program. However, DART suffers from a high number of duplicated paths due to the random process of generating the program paths, which leads to producing the same path more than once.

Forms of early works (for example, the Concolic Unit Testing Engine (CUTE) [24] and execution generated executions (EXE) [25]) consider the system environment in the assessment by implementing external calls that make use of real and concrete arguments, which constrains the behaviors that they could delve into as compared to an entirely symbolic strategy, which could be impracticable. In the context of an online executor, this selection can lead to having external calls from distinct and well-defined paths of execution that inhibit each other.

Williams et al. [26] suggested a prototype PathCrawler tool for automated test case creation to meet the paths requirement. It begins with source code instrumentation in order to determine the symbolic execution sequence whenever the code being tested is executed.

Subsequently, the instrumented code corresponding to the test scenario is executed until a complete feasible path set has been assessed. However, the tool faces an explosion of paths created due to extensive combinations.

Sen et al. [6] suggested MultiSE, where symbolic execution comprises incremental merging of the states without the use of auxiliary variables. The fundamental concept behind MultiSE is based on a different state representation, where all variables are mapped to guarded symbolic terms, referred to as a value summary. They implemented their prototype for JavaScript using the Jalangi framework [27]. However, MultiSE treats each condition separately, which increases the probability of producing duplicate paths, and it does not support web forms.

Cadar et al. [7] illustrated the LLVM compiler-based KLEE symbolic execution software [10], which can solve optimization problems and enhance performance by an order of magnitude and process several programs that may otherwise be intractable. Its search heuristics effectively select paths from large sets of paths to obtain high code coverage. KLEE is also a crucial constituent in several ventures and research such as Cloud9 [28], GKLEE [29], KLEENet [30] and Klover [31]. Nevertheless, the LLVM compiler is based on the algorithm formulated by Cytron et al. [32]. The compiler is similar to a non-SSA CFG and records the local parameters into memory, which typically are not in SSA form. The results indicated that about a quarter of the instructions created by the front end of the LLVM are in this format.

Havelund and Pressburger [8] suggested the Java PathFinder (JpF) translator, which translates Java to Promela, the modeling language required for using the Spin model checker. JpF transforms specified Java code to its corresponding Promela version, which may then be processed. The key drawback of this tool is that it cannot be utilized for substantiating any authentic Java application if no non-trivial quantity of work is completed by its user. For the resolution of this issue, Shafiei and Breugel [33] recommended an extension, *jpF-nhandler*, of JpF that enables automation of the handling of native methodologies. Automation of the interlinking for the execution of the native code and the model checking of Java code occurs. However, the extension lacks soundness and, as the author of [33] stated, it is not comprehensive.

Nguyen et al. [34] proposed a method (CFT4CUnit) for creating static direction-based test cases specific to C functions. Initially, a CFG is created as a C function that comprises the source code, including the inputs, maximum iteration count for loops, and the coverage conditions. Feasible test sequences are then determined using the backtracking technique, a Z3 satisfiability modulo theory (SMT) solver and symbolic execution for traversing the CFG. This technique's primary drawback is that it is time-intensive when addressing the constraints under a CFG possessing numerous decisive vertices but many fewer infeasible sequences. Nguyen et al. [12] proposed the SDART tool, which improved upon the DART [23] tool, along with their prior research [34] by integrating these techniques with a static test data-producing technique that could determine feasible paths using relatively fewer iterations. The proposed static analysis scheme creates several partial test paths that traverse unvisited branches. Subsequently, test data creation is attempted by traversing these test paths. Traversed branches are marked in order to prevent revisiting the branches more than once.

Table 1 summarizes the most popular symbolic execution studies that aim to detect feasible paths, the methods and tools that their approaches are based on, the limitations of their studies and the programming language used to implement and evaluate their approaches.

**Table 1.** Symbolic execution studies to detect feasible paths.

Name	Technique	Challenges	Programming Language
DART [23]	Static source code parsing, random testing and dynamic analysis	It devolves to random testing when pointers are encountered and can only handle integer constraints.	C
EXE [25]	Automatically generating test cases for the codes on its own	Due to the use of the STP [35] solver, some paths are missed because the solver does not accurately handle all operations.	C
CUTE [24]	Concrete and symbolic execution	Failure to produce test inputs for practicable program paths.	C
PathCrawler [26]	Combining static analysis and dynamic analysis	A combinatorial explosion in the number of execution paths	C
MultiSE [6]	Merging states incrementally during symbolic execution	Treat each condition separately, which increases the chance of producing duplicated paths	JavaScript
KLEE [7]	Cytron et al. [32] algorithm and symbolic execution	Consumes a lot of time when producing feasible paths. JpF cannot be utilized for substantiating any Java application if no non-trivial quantity of work is completed by its user.	C
JpF [8]	Java virtual machine (JVM) and runtime scheduler	It lacks soundness and it is not comprehensive	Java
jpf-nhandler [33]	Automatically delegating the execution of the native method	Solving the constraints' expressions proves to be highly time-intensive; moreover, some feasible paths are missed.	Java
CFT4Cpp [34]	Backtracking algorithm, symbolic execution, and Z3 solver	Some feasible paths that contain multiple conditions in the same branches are missed	C
SDART [12]	Combined DART [23] with boundary values of input parameters		C

To overcome the limitations of previous studies in detecting feasible paths, our proposed approach aims to decrease the time required to detect feasible paths, which was the main issue in the previous studies that used the LLVM compiler [7,28–31]. Some of the previous studies [23–25] also missed some feasible paths because of the use of the Simple Theorem Prover (STP). Therefore, the Z3 solver [36], as a high-performance theorem prover, could be used to solve different complex theorems [37]. In addition, we present a method to avoid generating duplicated feasible paths by calculating the weight of each path and avoiding the repetition of other paths with the same weight. Based on our knowledge, there is no available study that has been conducted to detect feasible paths in PHP. Therefore, we implemented and evaluated our approach in a PHP environment, which is considered the most common web technology with which to build a web application.

#### 4. Detection Methodology

This section presents our proposed approach and algorithm design, including (i) converting the source code to minimal SSA form; (ii) symbolically executing the program with a constraints extractor; (iii) avoiding repeated detection of the same feasible paths among the program paths; and (iv) solving the constraints of each new path.

In this method, the program under test is parsed and converted to an abstract syntax tree (AST). Then, a direct translation from the AST into an SSA-based intermediate representation, which includes optimization of the SSA form to be pruned and the minimal SSA form, is conducted. The second stage starts by assigning symbolic inputs for each superglobal variable (the user input variables) and the undefined variables to represent the variables with unknown values that might affect the path conditions. The path generator algorithm will traverse each block in the SSA form. In each generation, it will flag each

path not to be detected again and ensure that all paths were detected once. During the path generation, the conditions inside each path are extracted and stored under that path. The results of path generation and condition extraction are checked again by the introduced method (path weight method) to calculate each path’s weight, and only the paths that have a unique weight are passed to the constraint solver to check their satisfiability (feasible or not) under the given inputs. Figure 4 shows the full steps of our approach to detect the feasible paths in a PHP source code.

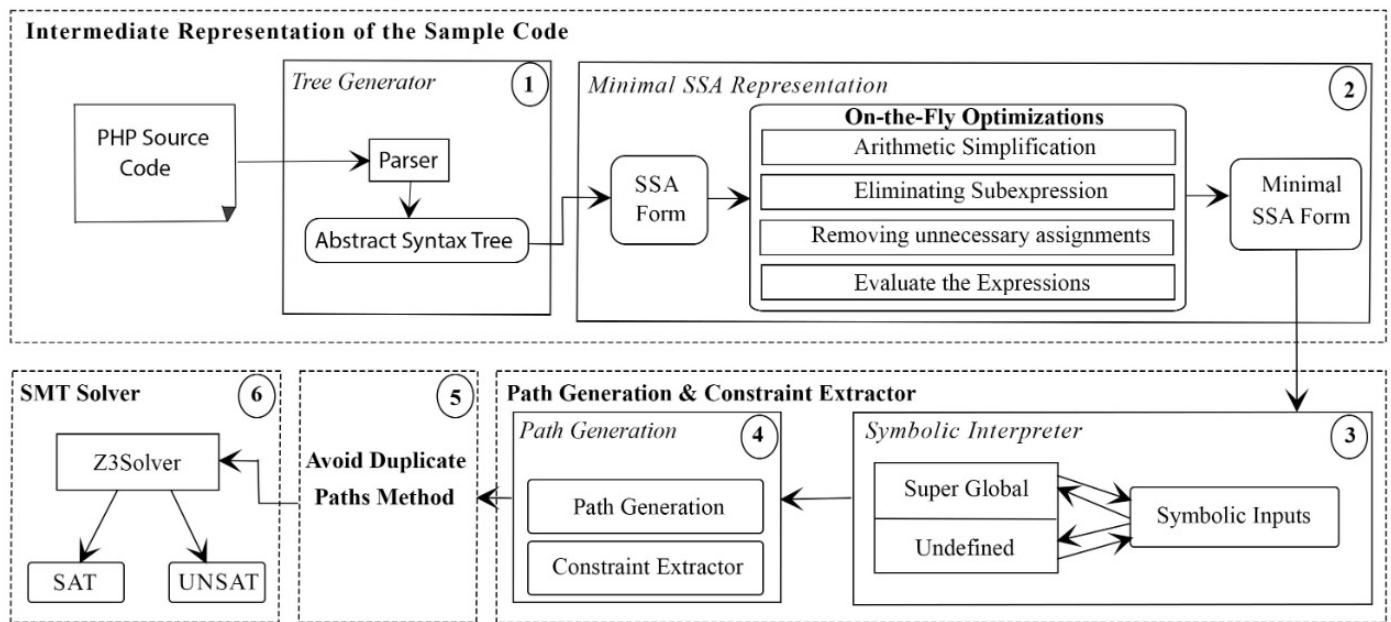


Figure 4. The framework of the proposed scheme.

#### 4.1. Parser

The PHP source code should be changed to an intermediate representation, such that it can be converted to SSA format afterward. The structure of the program is indicated by AST; hence, a grammatical and lexical analysis must be performed initially for the source code. In the context of this research, PHP-Parser [38] was employed for the grammatical and lexical analysis. PHP-Parser produces an AST, which helps immensely with PHP static analysis.

#### 4.2. Minimal and Pruned SSA form

The SSA [39] form is a property of intermediate representations. This property ensures that each variable is defined and written only once. As a result, programs in SSA form encode explicit data flow relations. Figure 5 illustrates an example of the SSA form and  $\phi$  function.

To understand the reasoning behind the SSA representation, a straight-line code is an appropriate way to begin. Figure 5 shows that a unique name is provided for every assignment made to a variable, and wherever the assignment is used, it is renamed to a new assignment name. The majority of programs comprise join and branch nodes. A unique assignment form referred to as the phi function ( $\phi$ ) was used at the join nodes. The operands of the  $\phi$  function represent the assignments to  $V$  that reach the join. Any further use of  $Y$  is considered a use of  $V_5$ . The previous variable  $Y$  is replaced with new variables  $V_3, V_4,$  and  $V_5$ . All uses of  $V_i$  are arrived upon by a single assignment to  $V_i$ . A single assignment to  $V_i$  can be found in the complete code, which simplifies the record maintenance process for numerous optimizations.



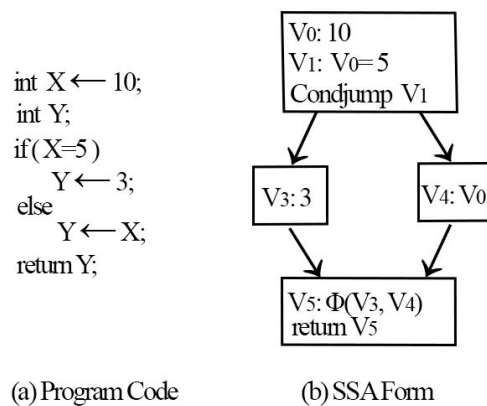


Figure 5. An example of program code converted to SSA form.

In this paper, we use the algorithm from Braun et al. [11] to generate the minimal SSA form, where they used on-the-fly optimizations with an SSA compiler to decrease the number of  $\phi$  functions. Initially, a mathematical simplification is employed where the constructors pertaining to the IR node are subject to peephole optimizations and output simpler nodes wherever feasible. Considering an example, the mathematical expression  $X-X$  always amounts to zero. Furthermore, common subexpression removal, reuse of the present values identified using local values, and an evaluation of the constants during compilation are performed—for example,  $2 \times 3$  may be optimized as 6. Finally, the local variables may have unnecessary arguments that must be removed ( $X = Y$  is one such example). The SSA form does not require assignments of this form; hence, it is feasible to use the right-hand side value directly. The effectiveness of such optimizations is depicted in Figure 6.

The minimal static single assignment (MSSA) form before the application of on-the-fly optimizations is depicted in Figure 6b. When the optimizations are enabled, the initial difference is observed when value  $V_1$  is constructed. Here, any comparison with zero will return false; therefore, the value is simpler during arithmetic simplification. During the next step, constant propagation converts a comparison with zero returning false. Now, the state at the jump condition is false; therefore, it is feasible to omit the code inside the “then” block. Concerning the “else” block, copy propagation is conducted by assigning  $V_0$  to  $Y$ . Along the same lines,  $V_5$  vanishes and, ultimately, the code returns  $V_0$ . The optimized SSA form of the code is illustrated in Figure 6c. It may be observed from the example that the run-time optimizations suggested by Braun et al. [11] may potentially reduce the instruction count and  $\phi$  functions in the SSA form. After finding the minimal SSA form of the PHP source code, the next section describes the symbolic execution stage.

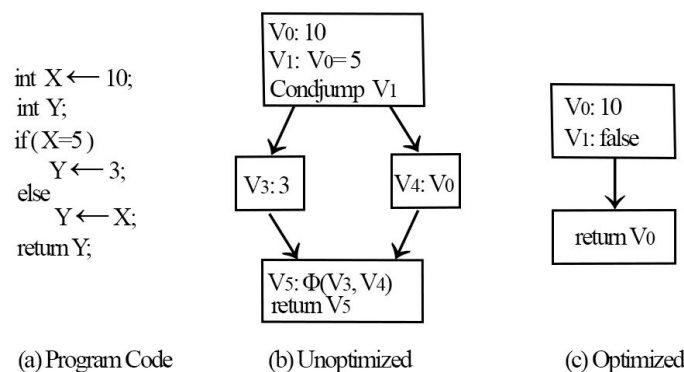


Figure 6. Converting a program code to Minimal SSA form. (a) Program code. (b) SSA form before on-the-fly optimizations. (c) Optimized SSA form of the code.

### 4.3. Symbolic Interpreter

During the generation of the SSA form, and once any superglobal variable or any uninitialized variable is detected, we assign a symbolic input for that variable. PHP has several in-built variables called superglobal variables. These variables store the input from the external user, which is unpredictable. Hence, the values are kept symbolic. In the scope of this study, all superglobal variables of the PHP language are considered [40]. A full symbolic execution assignment algorithm for the symbolic interpreter is shown in Algorithm 1.

---

#### Algorithm 1: Symbolic Interpreter Algorithm.

---

```

input :SSA form
output:SSA form with symbolic inputs
SGV [] ← list of all super global variable in PHP
SGD [] ← list of all super global variables detected
CBV [] ← list of the current block variables
while variable ∈ MSSA do
  if variable is NULL then
    // uninitialized/unknown variables
    CBV ← symbolic input
  else if variable in SGV then
    // super global variable
    if variable in SGD then
      // if defined before
      CBV ← SGD[variable]
    else
      CBV ← symbolic input
      SGD ← symbolic input
    endif
  endif
endif
endwhile

```

---

For instance, “\$name = \$\_POST[‘username’]” points to the value inputted by the external user using the POST technique; hence, the value is not known. Consequently, “\$name” is allotted the symbolic value “postsymbolicusername”. Once the symbolic value is assigned for the superglobal variable, this variable is stored in an array so that we can notice if the same superglobal variable is assigned for another variable. Meanwhile, if we detect the use of a variable that has not been defined before, we assign a symbolic value “undefinedsymbolic” for that variable. The complete process of assigning symbolic values to superglobal variables and uninitialized variables is presented in Algorithm 1.

To solve the loop heuristically, we provide a symbolic input for every loop condition variable present in the loop that is being assessed (loop condition), which allows the loop to be evaluated symbolically under the given inputs. Figure 7 illustrates the previous example code in Figure 1 with a while loop; this shows the assignment of a symbolic input for the loop condition to be executed symbolically.

The loop in Figure 7 is defined at Line 6 with the condition ( $i < 3$ ), which means the loop iteration will continue to execute until the condition terminates. In the loop body, another condition ( $a > b \ || \ a < c$ ) is defined inside the “if” statement at Line 7, which it will execute in each iteration of the loop (3 times during every run of the example program). To avoid the repetition of the loop iteration, a symbolic input ( $S$ ) was assigned for the loop condition ( $i < 3$ ). Afterwards, the symbolic loop condition ( $i < S$ ) was added for each path inside the loop, and the logical operator “AND” (&&) was added to ensure that the path condition of the loop was satisfied and executed at least once.

<pre> 1  &lt;?php 2  \$a = 15; 3  \$b = 10; 4  \$c = 25; 5  \$i = 0; 6  while (\$i &lt; 3) { 7      if(\$a &gt; \$b    \$a ≤ \$c) 8          { echo \$a; } 9      else 10         { echo \$c; } 11         \$i++; 12     } </pre>	<p><b>Path Conditions:</b></p> <p>P1) (True: \$i &lt; S) &amp;&amp; ((True: \$a &gt; \$b)    (True: \$a ≤ \$c))</p> <p>P2) (True: \$i &lt; S) &amp;&amp; ((True: \$a &gt; \$b)    (False: \$a ≤ \$c))</p> <p>P3) (True: \$i &lt; S) &amp;&amp; ((False: \$a &gt; \$b)    (True: \$a ≤ \$c))</p> <p>P4) (True: \$i &lt; S) &amp;&amp; ((False: \$a &gt; \$b)    (False: \$a ≤ \$c))</p> <p><b>Total Paths:</b></p> <p>P1) 1-2-3-4-5-6-7-8-11 (Feasible Path)</p> <p>P2) 1-2-3-4-5-6-7-8-11 (Duplicated Feasible Path)</p> <p>P3) 1-2-3-4-5-6-7-8-11 (Duplicated Feasible Path)</p> <p>P4) 1-2-3-4-5-6-7-10-11 (Infeasible Path)</p>
---	--

**Figure 7.** Assigning a symbolic inputs for the loop condition to be executed symbolically.

Since our approach focused on detecting new feasible paths and avoiding duplicated paths, rather than keeping the loop traverse for n number of iterations in the next stage (Section 4.4), our approach prepared the loop with symbolic inputs. This helped with path generation and constrained the extractor algorithm to connect the symbolic loop condition with the path conditions inside the loop body.

#### 4.4. Path Generation and Constraints Extractor

Each block in the SSA form contains some instructions (i.e., variables, expressions, and conditions) of that block. As the SSA form contains one start block and one end block, it helps to know the start and end points of each path. The complete process of generating the paths and extracting the conditions from the minimal SSA form is presented in Algorithm 2.

The path generator always starts from the first block as an initial block. It will first store the instructions of each block in an array of instructions and those to be used later in the constraint extractor to determine the conditions' side values. The functions inside the SSA structure are defined as a separate group of blocks. Once the generator finds a function call inside the block, it will move to the function blocks and track the instructions inside that function, and each instruction inside the function is stored in the array of instructions for that path, and the conditions inside that function are also stored in the path conditions. The last block in the function block contains the termination of that function, which will return the generator to the function call instruction.

Each condition contains two or more children based on the condition types (i.e., if and switch). If the generator finds a condition, the generator will choose one of the children it has not yet visited, and the chosen child will be marked as visited. In this way, the generator ensures that it will not follow the same path in the next generation. Each child block will be visited once during the path, and each condition state among the path is stored in the array of conditions. At the termination of the path, a collection of conditions exist in the array of the conditions, and the path generator starts from the first block again to generate a new path.

Since all variables from each block are stored in the array of instructions, the constraint extractor replaces the conditions of both side variables with their concrete values from the array of instructions. For example, a variable \$X was defined with a value of 5, followed by a condition (\$X == 5). The condition has left and right values that must be compared. First, the variable is stored in the array of instructions, and the constraint extractor searches for the condition of the left-side variable in the array of instructions and replaces the variable "\$X" with its concrete value, which is 5. The condition's right-side variable is a concrete value and, thus, it will not be replaced because it does not exist in the array of instructions for that path. The results of the path generator and constraint extractor are a list of path

conditions and their instructions are sent to the Z3 solver to check the satisfiability of these conditions.

---

**Algorithm 2:** Path Generation and Constraints Extractor Algorithm.

---

```

input :SSA form
output:Path conditions of the source code


---


Paths [] ← list represent the generated paths
Cond [] ← list represent the generated conditions
Instr [] ← list represent the generated instructions
i ← 1
Function PathGeneration(SSA):
  foreach B_i ∈ SSA do
    if B_i[type] is conditional block then
      if this conditional block Not Visited before then
        Replace condition variables with concrete values
        Cond ← Store the condition
        Flag this condition state as visited
        i ← Store the next block number
        PathGeneration(SSA)
      else
        | Move to next Block
      endif
    else if B_i[type] is Function then
      | i ← Store the Function block number PathGeneration(SSA)
    else if B_i[type] is Terminate then
      | Paths ← Instr & Cond
      | i ← 1
      | PathGeneration(SSA)
    else
      | Instr ← SSA[B_i][type]
    endif
  endforeach
endFunction

```

---

#### 4.5. Avoid Duplicated Feasible Paths

Duplicated paths are related to the branches that contain multiple conditions separated by logical operators (and, or, &&, and ||), as shown in Figure 1. These branches are split into multiple branches depending on the number of conditions inside them, which gives rise to the possibility of producing numerous paths under the same branches (called duplicated paths). The effects of not addressing these paths are manifested when a sensitive path flow is used to detect a threat in the source code. Duplicating the paths reflects an increase in false alarms in the results (detecting multiple threats in the code whilst all of them are related to one threat). To avoid producing duplicated paths, we introduce a path weight method (PWM) that can avoid generating duplicated paths.

The developed PWM can be implemented in five steps, as follows:

- Step 1: Assign a unique number to each normal block (non-conditional block) (NB) in the SSA form.
- Step 2: Count the sum of the NBs assigned a number in the current path and denote it as SNB, where SNB refers to the sum of all NBs in the current path.
- Step 3: Count the total number of conditional blocks in current path P (denoted as CB).
- Step 4: Count the total number of logical operators in current path P (denoted as LO).
- Step 5: Calculate the current path weight W(p) by using the following equation:

$$W(p) = \frac{SNB}{CB - LO} \quad (1)$$

The calculation of the current path weight in Equation (1) is performed by dividing the sum of all blocks' numbers (SNB) by the difference between the total number of conditional blocks (CB) and the total number of logical operators (LOs) in the current path. The difference between CBs and LOs provides the actual number of CBs without any repetition. The sum of all normal blocks (SNB) is used to give diversity to the weight for each path. Therefore, we can ensure that each path has its own weight, and the same weight is given to the duplicated paths. The PWM's steps for checking duplicated paths and avoiding them are presented in Algorithm 3.

---

**Algorithm 3:** Avoid Duplicated Paths Algorithm.

---

```

input : All generated paths
output: Only new feasible paths (non-duplicated)
GP [] ← list of all generated paths
NFP [] ← list of new paths (non-duplicated)
PW ← 0 // is the path weight
while Path ∈ GP do
  SNB ← Sum of all blocks unique number in Path (except conditional blocks)
  CB ← Count the conditional blocks in Path
  LO ← Count the logical operators in Path
  PW ← (BS)/(CB+LO)
  if PW(path) in NFP then
    | Path ← Duplicated path
  else
    | Path result ← Check the satisfiability of the Path using Z3 solver
    | if Path result is feasible then
    | | NFP ← PW(Path)
    | else
    | | skip and continue to the next path
    | endif
  endif
endwhile

```

---

Figure 8 shows the results of the path weight method for the previous example code in Figure 1. If the path weight has not been detected before (new path), then our approach will proceed to the next step to check its feasibility (such as P1). If our PWM finds a path that has the same weight as that of a path that has been detected previously as a feasible path (such as P2 and P3, which have the same weight ('7') as P1), then it will not proceed to the next stage, in which its feasibility is checked, and the path will be considered a duplicated path. However, if our PWM finds a path that has the same weight as that of a path detected previously as an infeasible path (such as P4), then it will proceed to the next stage, in which its feasibility is checked, since the path was not detected as a feasible path before, and there is a chance for the SMT solver to solve these path conditions and consider it as a feasible path.

```

MINGW64:/c:/xampp/htdocs/DeFP
$ php run.php C:/xampp/htdocs/DeFP/TestProgram/

(assert (or (= (> 15 10) true)=( < 15 25) true)))
sat
Weight = 7
New Path
-----
(assert (or (= (> 15 10) true)=( < 15 25) false)))
Weight = 7
Duplicated Path
-----
(assert (or (= (> 15 10) false)=( < 15 25) true)))
Weight = 7
Duplicated Path
-----
(assert (or (= (> 15 10) false)=( < 15 25) false)))
unsat
-----

<<<< Results >>>>
+-----+-----+-----+-----+
| File   | #I | #B | #FPD | PGT(s) |
+-----+-----+-----+-----+
| test.php | 16 | 2 | 1 | 0.004 |
+-----+-----+-----+-----+

```

**Figure 8.** The satisfiability results of the program in Figure 1 using Z3 solver.

#### 4.6. Z3 Solver

The Z3 solver is based on the satisfiability modulo theory and was formulated by L. de Moura and N. Bjorner while they were working at Microsoft Research [36]. The SMT solver works in such a way that when a first-order logic formula is specified, the solver determines if satisfiability is met. Formula  $f$  is considered satisfiable if a value that leads to  $f$  being true exists. The solver is given inputs regarding the expression and the set of applicable constraints. The solver then attempts to identify a solution applicable under the specified constraints.

The reason behind using Z3 was that it has been established as a high-performance technique for theorem proof [41]. Additionally, this solver is available for free under the Microsoft Research License Agreement (MSR-LA). Furthermore, it has Application Programming Interfaces (API)s for several languages, which make this tool compatible with several platforms.

Once the path generation produces the condition of each path, it transforms them to the Z3 context based on their guidelines [36], and the results are sent to the Z3 solver to check satisfiability of those path constraints. Figure 8 shows the results of our proposed work on the example program in Figure 1.

The first column in the result table in Figure 8 shows the file name. The second column shows the number of instructions produced by the minimal SSA algorithm. The third column provides the number of branches analyzed, and the detected feasible paths (new paths) are shown on the fourth column. The last column presents the time (in seconds) required to generate all of the paths, excluding the Z3 solver time. Figure 8 shows that the first path was checked, and the results indicated that it was satisfiable (sat) under their given inputs. The second and third paths were not checked by Z3solver since they have the same weight as the first path, which was detected as feasible. The fourth path was not satisfied (unsat) by the Z3 solver on the basis of the given inputs. Therefore, it was considered an infeasible path and was not executed at all under the given inputs.

### 5. Experimental Results and Evaluation

The proposed approach was implemented in PHP, and the implementation followed the architecture in Figure 4 and the approach presented in the previous sections. The

objective of the experimental evaluation was to answer the following questions: (1) Is the proposed approach able to detect the feasible paths and avoid the detection of duplicated paths amongst the test programs (Section 5.3)? (2) Does the minimal SSA form reduce the time required to generate the paths compared with other studies that used the SSA form (Section 5.4)? (3) Is the proposed approach able to process a large set of PHP applications (Section 5.5)?

### 5.1. The Dataset

The proposed approach focuses on PHP as the most popular web technology for building a web application. To our knowledge, no study has been conducted to detect feasible paths on the basis of PHP. Therefore, we used the same test programs written in C language that were used in related studies [42–51]. These programs were rewritten in PHP. Table 2 presents the characteristics of the test programs, including the number of lines of codes, LoC, the number of branches of each program, B (can be either selection or loop statements, such as IF, IF ELSE, WHILE, FOR and DO), Description, the number of feasible paths (non-duplicated paths), F, and their reference(s), Refs.

**Table 2.** Test programs.

Program	LoC	B	Description	F	Refs.
eR1985	18	3	(Expint) Raises one integer to the power of the other.	5	[42–44]
fcB2002	21	4	(Floatcomp) Compares three floating point numbers and has some selections.	5	[43–47]
gA2008	16	3	Finds the greatest common divisor between any given two integers.	5	[43,44,48,49]
rA2008	16	3	Finds the remainder in integer division.	4	[43,44,46,48,50,51]
tA2008	14	3	Determines whether three given numbers that represent three lengths on a plane form a scalene, isosceles, equilateral, or not a triangle.	4	[43,44,46–48,51]
tM2004	20	4	(Triangle) Classifies three numbers representing triangle side lengths into five type triangles: scalene, isosceles, right, iso-right, or equilateral.	7	[43,44,47]
ttB2002	38	7	(Tritype) Accepts three integers representing sides of a triangle, classifies its type, and computes its area.	8	[43,44,47]

LoC: Lines of Codes, B: the number of branches of each program, Description: a description of each program, F: the number of feasible paths (non-duplicated paths), Refs: reference(s) of each program.

The programming language used to write these programs does not affect the results of the programs because target paths are based on the control flow graph; thus, the program does not have a specific statement related to one programming language [43]. This task is a common practice in testing experiments undertaken by several studies that rewrote these programs in MATLAB [44,48]. Others rewrote them in Java [52–54]. The source code of each program in C and PHP is publicly available (<https://github.com/Amarashdeh/FP-PHP-C-TestPrograms>, accessed on 1 April 2021).

The entire experiment was conducted on the operating platform Windows 10, 8 GB RAM, Intel i5-7200U CPU. PHP version 7.4 was used to build the proposed approach with XAMPP local server version 7.4.

### 5.2. Performance Evaluation Metrics

The proposed approach started by analyzing the source code to generate an AST. Then, the AST was converted to a minimal SSA form, which is the form that helps to reduce the instructions in the normal SSA form. Two well-defined algorithms were used for the symbolic inputs' assignments and the path generation including the condition extractor. The entire process of generating the feasible paths was conducted automatically, which means that no human action is needed for generating the path conditions. A path weight method was proposed to avoid detecting duplicated feasible paths. The last stage

checks the paths' conditions by using the Z3 solver. The results of the Z3 solver show the satisfiability of each path and the values of each symbolic input that could satisfy the condition if it is a feasible path.

The proposed approach's performance was assessed based on the number of feasible paths detected, the number of duplicated paths detected, and the time required to generate paths. "Recall" measures the percentage between the number of feasible paths correctly detected and the actual number of feasible paths existing in the source code. "DuplicatedPathsPercentage" measures the percentage between the number of duplicated paths detected and the total number of paths detected in the program. The precision and recall rates were calculated with Equations (2) and (3), respectively.

$$Recall = \frac{F_{Detected} - F_{False}}{F_{True}} \quad (2)$$

$$DuplicatedPathsPercentage = \frac{DF_{Detected}}{F_{Detected}}, \quad (3)$$

where  $F_{Detected}$  denotes the number of feasible paths detected and  $F_{True}$  denotes the number of feasible paths that exist in the detected program;  $F_{False}$  denotes the number of feasible paths detected that are infeasible in reality;  $DF_{Detected}$  denotes the number of duplicated feasible paths detected.

The evaluation of the time required to generate the paths was performed using the PHP built-in function `microtime()`, which is a function that returns the current Unix timestamp in microseconds [55]. By default, `microtime()` yields a string in the form "msec sec," where sec is the number of seconds and msec gauges microseconds that have elapsed since sec, which is also stated in seconds. We set up the function at the first stage of generating the SSA form and it ends at the termination of generating the paths.

### 5.3. Feasible Paths Detection

Table 3 depicts the outcomes of our approach and those of related studies. We compared the proposed approach with the most common engines and tools that are publicly available, such as KLEE [7] and SDART [12]. The KLEE engine is the most related work to ours. It uses the SSA form in the LLVM compiler with symbolic execution to detect feasible paths. We selected the latest version of KLEE 2.2, which was released on 7 December 2020. The SDART tool is based on CFT4CUnit [34] and DART [23]. It reduces the number of test cases required to generate feasible paths compared with the previously mentioned tools [23,34]. The second column in Table 3 shows the real number of feasible paths (non-duplicated paths) in each program (denoted as #TFP). The third and fourth columns show the KLEE results, where #FD denotes the number of feasible paths detected, and #DFD denotes the number of duplicated feasible paths detected. The fifth and sixth columns show the results of SDART, followed by the proposed approach's results in the seventh and eighth columns.

Table 3 presents the results of each study on detecting feasible and duplicated paths. Each feasible path generated by each tool was checked to determine whether it is a new path or duplicated from previous detected paths. KLEE detected 45 feasible paths, 10 of which were duplicated. This result means that the number of new feasible paths detected by the KLEE engine was 35. Meanwhile, the SDART tool detected 27 feasible paths, four of which were duplicated. Notably, the four paths were found in the test programs that had more than two conditions at the same branch. The number of new feasible paths detected by SDART was 23. Our approach detected 35 feasible paths without any duplicated paths amongst the results, and all the duplicated paths were addressed and avoided before checking their feasibility.



**Table 3.** Results of detected feasible paths and duplicated paths.

Program	#TFP	KLEE		SDART		Our Approach	
		#FD	#DFD	#FD	#DFD	#FD	#DFD
eR1985	5	7	3	1	0	4	0
fcB2002	5	6	1	5	1	5	0
gA2008	5	4	0	2	0	4	0
rA2008	4	3	0	3	0	3	0
tA2008	4	8	4	6	2	4	0
tM2004	7	8	1	4	0	7	0
ttB2002	8	9	1	6	1	8	0

#TFP: the real number of feasible paths (non-duplicated paths) in each program, #FD: the number of feasible paths detected, #DFD: the number of duplicated feasible paths detected.

To answer the first question, we calculated the recall and duplicated path percentages for each study result by using the previously provided metrics (2) and (3). Table 4 shows the recall percentage of each study.

**Table 4.** Summary of KLEE, SDART, and our approach's results.

Metric	KLEE	SDART	Our Approach
Recall	92.1%	60.5%	92.1%
Duplicated paths	26.6%	14.8%	0%

The results of analyzing the test programs in Table 3 indicate that the proposed approach and KLEE have the same recall percentage (92.1%). The recall rate can reflect the comprehensiveness of the test results. Both approaches adopted the same structure to detect feasible paths. However, our approach reduced the number of instructions in the SSA form by using the minimal SSA form [11]. Meanwhile, KLEE detected 10 duplicated paths that were detected in the previous test generation. By contrast, our PWM successfully addressed such paths and did not proceed to check their satisfiability in the Z3 solver. SDART's recall percentage (65.7%) for detecting feasible paths was the lowest in the other studies. The test generation in SDART had an issue in covering the branches that had an infinite loop (fcB2002); it generated 45 test cases, without the ability to cover the last path that is related to the "while" loop. SDART missed 15 feasible paths that were not detected by the test programs. However, SDART was better than KLEE at reducing the number of duplicated paths in the test generation (duplicated path percentage), and the usage of this method shows its efficiency in avoiding duplicated paths under two conditions at the same branch.

#### 5.4. Paths Generation Time

To answer the second question, we compared the proposed approach with KLEE to reflect the reduction in the number of instructions analyzed by the minimal SSA form. KLEE [7] was implemented as a virtual machine for LLVM. However, the LLVM compiler uses the algorithm of Cytron et al. [32] to produce the SSA form. The algorithm that we used to generate the minimal SSA form by Braun et al. [11] reduced the number of instructions in the program. Table 5 shows the number of instructions (#I) and the time required for each program to generate its paths (T) in seconds. The time shown in Table 5 excludes the SMT solver's time to check the satisfiability of each path because many studies have already been conducted to compare the SMT solver's time and performance [56–58].

**Table 5.** Comparison of KLEE (using SSA) and the proposed approach's (using minimal SSA) times for generating paths among the test programs.

Program	KLEE		Our Approach	
	#I	T(s)	#I	T(s)
eR1985	66	0.014	53	0.007
fcB2002	70	0.007	55	0.006
gA2008	52	0.009	41	0.004
rA2008	52	0.005	39	0.004
tA2008	92	0.016	73	0.012
tM2004	53	0.004	49	0.003
ttB2002	190	0.433	155	0.293

#I: the number of SSA instructions generated in each program, T(s): the time required for each program to generate its paths in seconds.

The comparison of the number of constructed instructions (#I) indicates that the proposed approach generated fewer constructions than KLEE. The number of instructions generated by the programs depends on the LoC of each program and the number of conditions inside each branch. For example, the instructions in programs that have multiple conditions at the same branch (such as ftB2002, tA2008, and ttB2002) are more numerous than those in programs that have one condition inside the same branch. In addition, the results in Table 4 indicate that the proposed approach reduced the time for generating the paths. The last program (ttB2002) had more branches than the other programs. It had seven branches, five of which contained at least more than two conditions at the same branch. LLVM generated 190 instructions, and KLEE traversed all of these instructions to generate paths in 0.433 s. Meanwhile, our proposed approach generated 155 instructions under the same program and consumed a total of 0.293 s to generate the paths. From these results, we can assume that the proposed approach is efficient at reducing the time required to generate paths.

### 5.5. Large Scale Programs

Our proposed approach is evaluated with a large set of test cases. This evaluation is carried out to prove that the proposed approach is capable of detecting feasible paths without any duplications from the PHP program. The proposed approach was implemented with 893,575 LoC in 10 WordPress plugins [59] as shown in Table 6, where the first column shows the plugins' names, with the plugin versions tested by each proposed approach in the second column. Column three shows the number of PHP files in each plugin, followed by the line of code (LoC) for each plugin. The number of constructed instructions and analyzed branches are shown in the fifth and sixth columns, respectively. The detected feasible paths (#FD) among each plugin are shown in the seventh column, followed by the detected duplicated feasible paths (#DFD). Lastly, the time required to generate the paths is shown in the last column. We deployed the Cloc (<https://github.com/AIDanial/cloc>; accessed on 14 April 2021) tool to count the number of PHP files and LoC in every plugin. Cloc has been utilized in recent studies for similar metrics [60–62].

Table 6 shows that the proposed approach detects, in total, 152,669 new feasible paths (Total #FD) within 646.06 s (Total #T(s)), where each path has its own weight (unique weight). It should be noted that the 10 programs consist of a number of strings or pure numerical comparison branches, which belong to data types that can be solved by a constraint solver. Therefore, it is possible to detect a good number of feasible paths using the proposed approach. With regard to the program, i.e., Gallery (a web application allowing users to publish and organize photos in albums), although it contains a large number of branches, the proportion of the constraints that could be solved was not high, owing to the limitations of the constraint solver. Therefore, the proposed approach only detected 2116 paths among the 3093 branches in the Gallery program. In addition, among the analyses of the 10 programs, it was noted that the proposed approach failed to analyze the built-in WordPress functions, i.e., `add_action("list_menu", function_name)`, which is

quite challenging when it comes to our approach, since it requires an understanding of the built structure of that function in WordPress that can be traversed and analyzed.

**Table 6.** Summary of running the proposed approach with open source applications.

Program	Version	Files	LoC	#I	#B	#FD	#DFD	T(s)
Peruggia	1.1	10	659	3664	59	235	462	1.86
Measureit	1.14	2	826	3939	34	158	79	0.97
Zippec	0.32	10	1160	2458	126	424	319	2.58
PHPLib	7.4	75	13,053	42,192	324	694	706	4.94
Getboo	1.04	160	21,318	119,526	1776	2977	1223	9.28
WordPress	2.0	215	30,147	205,816	3198	7345	4383	17.49
Gallery	2	586	83,787	184,533	3093	2116	1744	12.41
NeoBill	0.9	620	100,139	249,820	3365	11,740	9583	31.75
Phpmyadmin	2.6.3	287	143,171	625,406	2229	29,815	11,462	59.92
TikiWiki	21.4	1563	499,315	2,211,879	30,243	97,165	37,829	504.86

On the other hand, if a path is found to possess the same weight as a previous path that has been detected as a feasible path, it is considered a duplicated path. The proposed approach avoided 67,790 duplicated feasible paths (#DFD) among the 10 programs, all which were detected previously as feasible paths. The time required to analyze the instructions and to generate the paths was different in each program based on the number of instructions and conditions that were analyzed.

Since no study has thus far been conducted to detect the feasible paths in PHP, we followed the methods of various studies [17,63–65] to validate the large-scale results by randomly selecting 50 feasible paths from each plugin (detected by the proposed approach) and checking manually whether any of them were duplicated. Table 7 shows the results when checking for duplications in the detected feasible paths out of the 50 paths that were chosen randomly from each plugin. The first column (program) shows the plugin name, the second column (#FP) shows the number of paths that were chosen randomly for testing, and the third column (#DFP) shows the number of duplicated paths out of the 50 chosen paths.

The chosen feasible paths in Table 7 were analyzed and we checked whether any duplication existed with the detected feasible paths (i.e., the validation of these paths was implemented by utilizing inputs from Z3Solver to ensure the feasibility of the path). It can be noted that our work successfully accomplished our objective of producing no duplications in each presented program. This indicates that the proposed work was evaluated accordingly. A sample of the results is shown in Appendix A (test case of Peruggia).

**Table 7.** Manually validation of 50 detected feasible paths.

Program	#FP	#DFP
Peruggia	50	No duplication
Measureit	50	No duplication
Zippec	50	No duplication
PHPLib	50	No duplication
Getboo	50	No duplication
WordPress	50	No duplication
Gallery	50	No duplication
NeoBill	50	No duplication
Phpmyadmin	50	No duplication
TikiWiki	50	No duplication

## 6. Threats to Validity

While this research demonstrates the effects of minimal SSA representation and avoids duplicated paths in the generation of feasible paths, threats to the validity of certain results

remain that the reader should account for when interpreting any outcomes. These threats comprise internal, external, and construct validities, and this section clarifies such problems regarding valid study results.

### 6.1. Internal Validity

Internal validity threats concern mostly uncontrolled factors that can influence experimental results. The key internal validity threat here lies in probable faults arising in the execution of our strategy. To mitigate this problem, we reviewed our experimental scripts for different feasible path scenarios and cases to ensure correctness before carrying out all experiments.

### 6.2. Construct Validity

Threats to construct validity mainly involve the relationship between observation and theory, which arise mostly in regard to how performance is measured for the proposed method of generating feasible paths. Test programs were chosen for the evaluation. Moreover, the performance for each feasible path generation strategy was compared via objective metrics such as the amounts of feasible and duplicated paths detected and the time needed for generation.

### 6.3. External Validity

External validity involves the generalizability of the research findings. Could our approach be implemented with alternative languages effectively? The results indicate that reducing the number of instructions in the SSA form helps decrease the time required to generate the paths. In addition, the proposed method for avoiding duplicated paths has proven its efficiency in avoiding the detection of duplicated paths. In the future, we intend to experiment with various applications written in different languages, which entails creating a minimal SSA form in such languages. Then, the approach could be generalized to several other languages.

The proposed approach in PHP was compared with C-based approaches, where the number of feasible paths and duplicate paths are the same in both languages. The programming language used to write the 7 test programs does not affect the results of the programs because the target paths are based on the control flow graph; thus, the program does not have a specific statement related to one programming language [43]. However, it would have been better to test the time spent on implementing the paths in each language using the same programming language. The algorithm created by Braun et al. [11] was previously compared with that of Cytron et al. [32], and the results indicated that a non-optimized implementation of the algorithm of Braun et al. [11] is somewhat faster than Cytron et al.'s [32] algorithm. Therefore, using Braun et al. [11] algorithm, our results were significantly better than the results obtained using KLEE, as the Braun algorithm helped to reduce the number of instructions to be analyzed and the time required to generate the paths. However, to increase the confidence in the results, in the future, we aim to enhance the proposed approach with other algorithms and provide a full comparison based on PHP (as there are no currently available studies conducted on PHP).

## 7. Discussion

The proposed approach aims to strengthen studies on detecting feasible paths based on PHP. We selected PHP for two reasons. First, it is widely used in server side web application development, with PHP programs having been used in some 21 million online domains. Second, PHP has long been the center of prior research on the static detection of Internet vulnerabilities, and thus, it has readily available benchmarks. Therefore, the proposed approach would be useful for future studies by implementing their security vulnerability method on the detected feasible paths to help reduce the false positive rate in their results.

To our knowledge, no study has focused on detecting feasible paths in PHP. Thus, we compared our outcomes with related studies that used C language [7,66], and the results demonstrate the efficiency of our proposed approach in detecting feasible paths, avoiding the detection of duplicated paths, and reducing the time required to generate paths. Static analysis differs from dynamic analysis in that it can cover 100% of code lines. However, it cannot be conducted on multiple technologies such as PHP, Java, etc. Each approach focuses on one technology and analyzes that technology only.

KLEE [7] is a symbolic execution tool implemented as a virtual machine for LLVM. Nevertheless, the LLVM compiler uses the algorithm of Cytron et al. [32] while mimicking non-SSA by placing every local variable into memory, which is typically not in the SSA form. This method comes at the expense of expressing simple definitions and the use of such variables involving memory operations. Around 25% of all instructions shown to be generated by the LLVM front end can be categorized as such. These variables are eliminated by SSA construction immediately following IR construction. The algorithm that we used to generate the minimal SSA form, by Braun et al. [11], was compared with that of Cytron et al. [32], and the results indicated that a non-optimized implementation of the algorithm of Braun et al. [11] is somewhat faster than the heavily optimized implementation of Cytron et al.'s [32] algorithm within the LLVM compiler. The experimental results of the proposed approach indicate that our approach is more effective than KLEE [7] in terms of the time required to generate paths amongst program instructions. Furthermore, we introduced a method to avoid detecting duplicated feasible paths.

It is worth mentioning that the proposed approach could be implemented using other programming languages such as C or Java. However, first, this would require finding the minimal SSA representation of those languages by applying on-the-fly optimization to the original SSA form, in a similar manner to the algorithm by Braun et al. [11]. Then, the symbolic interpreter and path generation could be used to generate the path conditions. In addition, the proposed path weight method proved to be useful to address the duplicated paths and could be implemented using other programming languages to eliminate the duplicated paths.

The limitation of the proposed approach is that it focused only on PHP, because the nature of static analysis is that it can only analyze the source code of one technology at a time. The solution for this issue is to build the proposed approach based on those technologies or to propose an approach based on dynamic analysis. However, using dynamic analysis would mean that 100% of LoC cannot be covered, causing some missing paths to not be detected [67]. Furthermore, with such dynamic analytical approaches, it might not be possible to acquire a deeper understanding of how an application should behave, which may lead to lower rates of detection.

## 8. Conclusions

This paper proposed an approach to detect feasible paths based on minimal SSA representation and symbolic execution based on PHP. It starts by parsing the source code to present an AST, followed by converting the AST to minimal SSA form, which is later optimized to produce a minimal SSA form to decrease the number of instructions and the number of  $\phi$  functions that would be analyzed. A symbolic input for each superglobal variable and uninitialized variable is assigned, and the paths and the conditions among each path are generated. Furthermore, we introduced a method for avoiding detecting duplicated feasible paths. The path conditions were the target for the Z3 solver to check the satisfiability. To evaluate the proposed approach, we conducted experiments using seven test programs that have been used in the related studies and 10 large scale web applications. The results obtained indicated that the proposed approach improved the time required to generate the paths and avoided detecting duplicated feasible paths. For future works, we plan to implement the proposed approach based on another language such as C or Java. The proposed approach can be improved by implementing it on a multiprocessor or distributed systems, which it will help to decrease the time required to generate the

paths, especially for large systems. The proposed method (PWM) has the ability to avoid detection of duplicated feasible paths; thus, it could be combined with other evolutionary algorithms (i.e., as part of the fitness function in genetic algorithms) to reduce the number of iterations that will detect the same feasible paths more than once (duplicated paths). In addition, we will attempt to combine the proposed approach with taint analysis to detect security vulnerability in web applications. We expect the proposed approach to help decrease the false positive and false negative rates of static taint analysis. Moreover, the proposed approach could be extended to export the features of the source code, which could help machine learning algorithms to obtain knowledge on program flaws.

**Author Contributions:** Methodology, Z.F.Z. and K.S.; software, A.W.M.; formal analysis, A.W.M.; data curation, A.W.M.; writing—original draft preparation, A.W.M.; writing—review and editing, Z.F.Z. and K.S.; supervision, Z.F.Z. and K.S. All authors contributed substantially to the work reported. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Data Availability Statement:** The C and PHP source codes of each test program used in this study are publicly available at <https://github.com/Amarashdeh/FP-C-PHP-Datasets>; accessed on 15 May 2021.

**Acknowledgments:** The authors would like to thank Universiti Sains Malaysia and Arab Open University, Saudi Arabia for supporting this study.

**Conflicts of Interest:** The authors declare no conflict of interest.

## Abbreviations

The following abbreviations are used in this manuscript:

AST	Abstract Syntax Tree
API	Application Programming Interface
CUTE	Concolic Unit Testing Engine
CFG	Control Flow Graph
DART	Directed Automated Random Testing
DFD	Duplicated Feasible paths Detected
EXE	Execution Generated Executions
FD	Feasible Path Detected
PHP	Hypertext Preprocessor
IR	Intermediate Representation
Jpf	Java PathFinder
LoC	Line of Code
LLVM	Low-Level Virtual Machine
MSR-LA	Microsoft Research License Agreement
MSSA	Minimal Static Single Assignment
PWM	Path Weight Method
SMT	Satisfiability Modulo Theory
STP	Simple Theorem Prover
SSA	Static Single Assignment
TFP	True Feasible Paths

The following symbols are used in this manuscript:

$B$	Program branches
$CB$	The total number of Conditional Blocks in the current path
$CBV$	List of the current block variables
$DF_{Detected}$	The number of duplicated feasible paths detected
$F_{True}$	The number of feasible paths that exist in the detected program
$F_{False}$	The number of feasible paths detected that are infeasible in reality
$F_{Detected}$	The number of feasible paths detected
$GP$	List of all generated paths
$i$	Instructions
$LO$	The total number of Logical Operators in the path

<i>NB</i>	Normal Block (non-conditional block)
<i>NFP</i>	List of new feasible paths (non-duplicated)
<i>S</i>	Symbolic input
<i>SGD</i>	List of all super global variables detected
<i>SGV</i>	List of all super global variable in PHP
<i>SNB</i>	The sum of the normal blocks
<i>t</i>	Time
<i>TFP</i>	Real number of feasible paths in the program
<i>W(p)</i>	The calculated weight of the path

## Appendix A

This section shows samples of the results of the proposed approach on the Perugia 1.1 application.

### Appendix A.1

Figure A1 shows a feasible path that was correctly detected as a feasible path (not duplicated). The first part shows the symbolic variables of this path, followed by the conditions that were extracted from this path. The green font color shows the Z3solver results with a value for each symbolic variable that Z3solver suggests satisfies this condition. The last two lines show the path weight value (7.75) and the type of path (a new path that is not duplicated).

```

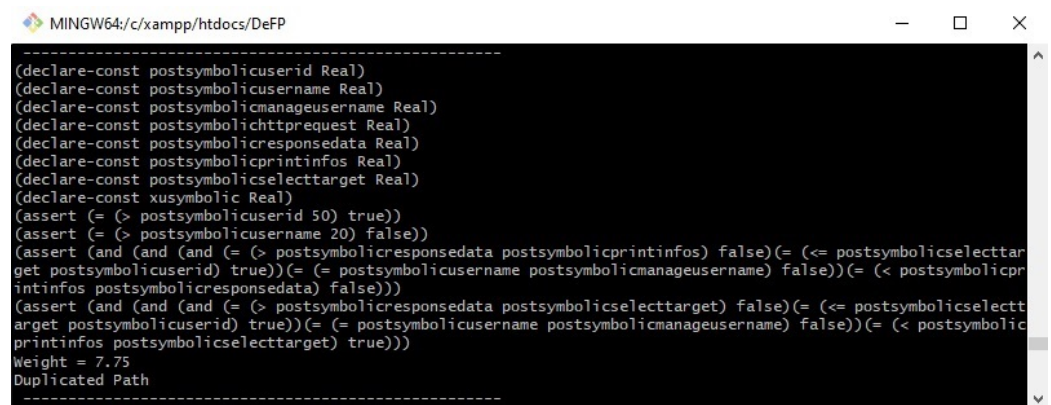
MINGW64:/c:/xampp/htdocs/DeFP
(declare-const postsymbolicuserid Real)
(declare-const postsymbolicusername Real)
(declare-const postsymbolicmanageusername Real)
(declare-const postsymbolichttprequest Real)
(declare-const postsymbolicresponsesdata Real)
(declare-const postsymbolicprintinfos Real)
(declare-const postsymbolicselecttarget Real)
(declare-const xusymbolic Real)
(assert (= (> postsymbolicuserid 50) true))
(assert (= (> postsymbolicusername 20) false))
(assert (and (and (and (= (> postsymbolicresponsesdata postsymbolicprintinfos) false) (= (<= postsymbolicselecttarget postsymbolicuserid) true)) (= postsymbolicusername postsymbolicmanageusername) true)) (= (< postsymbolicprintinfos postsymbolicresponsesdata) false)))
(assert (and (and (and (= (> postsymbolicresponsesdata postsymbolicselecttarget) true) (= (<= postsymbolicselecttarget postsymbolicuserid) true)) (= postsymbolicusername postsymbolicmanageusername) true)) (= (< postsymbolicprintinfos postsymbolicselecttarget) false)))
sat
(
  (define-fun postsymbolicprintinfos () Real
    1.0)
  (define-fun postsymbolicmanageusername () Real
    0.0)
  (define-fun postsymbolicresponsesdata () Real
    1.0)
  (define-fun postsymbolicusername () Real
    0.0)
  (define-fun postsymbolicuserid () Real
    51.0)
  (define-fun postsymbolicselecttarget () Real
    0.0)
  (define-fun postsymbolichttprequest () Real
    0.0)
  (define-fun xusymbolic () Real
    0.0)
)
Weight = 7.75
New Path

```

Figure A1. Example of feasible path detected in Perugia application.

### Appendix A.2

Figure A2 shows a duplicated feasible path that was detected as a duplicated path. The first section shows the symbolic variables of this path, followed by the conditions that were extracted from this path. Since the path weight was similar to a path that was detected previously (7.24), this path was considered as duplicated, and it did not pass to the next step (the feasibility check). The last line shows the path results for duplicated paths.



```

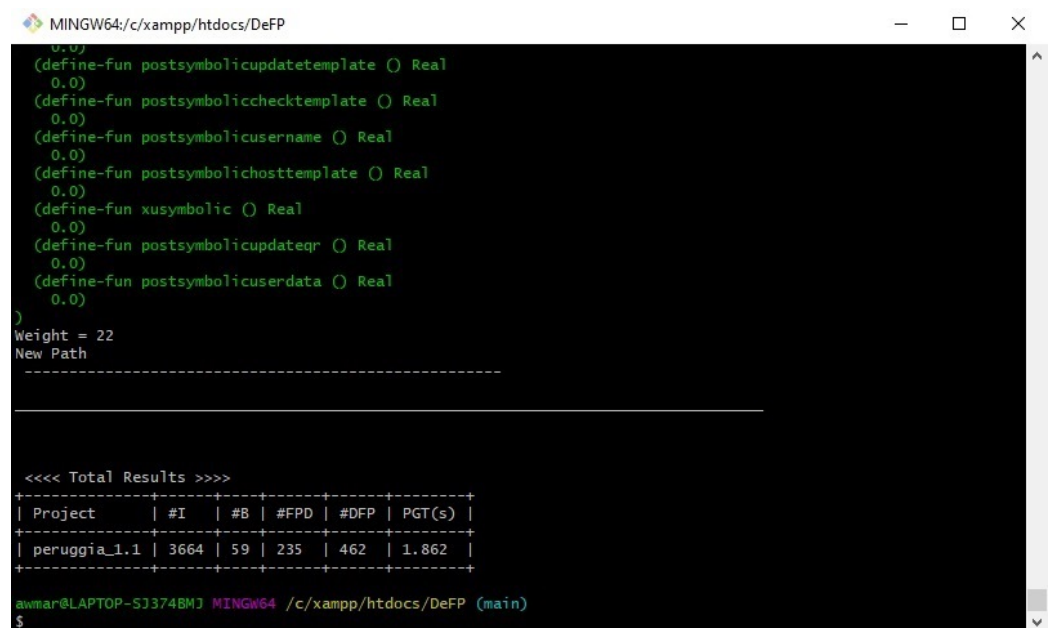
-----
(declare-const postsymbolicuserid Real)
(declare-const postsymbolicusername Real)
(declare-const postsymbolicmanageusername Real)
(declare-const postsymbolichttprequest Real)
(declare-const postsymbolicrespondedata Real)
(declare-const postsymbolicprintinfos Real)
(declare-const postsymbolicselecttarget Real)
(declare-const xusymbolic Real)
(assert (= (> postsymbolicuserid 50) true))
(assert (= (> postsymbolicusername 20) false))
(assert (and (and (and (= (> postsymbolicrespondedata postsymbolicprintinfos) false)=(= postsymbolicselecttarget postsymbolicuserid) true))(= postsymbolicusername postsymbolicmanageusername) false))(= (< postsymbolicprintinfos postsymbolicrespondedata) false))
(assert (and (and (and (= (> postsymbolicrespondedata postsymbolicselecttarget) false)=(= postsymbolicselecttarget postsymbolicuserid) true))(= postsymbolicusername postsymbolicmanageusername) false))(= (< postsymbolicprintinfos postsymbolicselecttarget) true))
Weight = 7.75
Duplicated Path
-----

```

Figure A2. Example of duplicated feasible path detected in Peruggia application.

### Appendix A.3

Figure A3 shows the final results of analyzing the Peruggia application, where 3664 instructions (I) and 59 branches (B) were analyzed. A total number of 235 feasible paths (FPD) was detected, while 462 duplicated feasible paths (DFP) were avoided within 1.862 s.



```

0.0)
(define-fun postsymbolicupdatetemplate () Real
  0.0)
(define-fun postsymbolicchecktemplate () Real
  0.0)
(define-fun postsymbolicusername () Real
  0.0)
(define-fun postsymbolicostemplate () Real
  0.0)
(define-fun xusymbolic () Real
  0.0)
(define-fun postsymbolicupdateqr () Real
  0.0)
(define-fun postsymbolicuserdata () Real
  0.0)
)
Weight = 22
New Path
-----

<<<< Total Results >>>>
+-----+
| Project | #I | #B | #FPD | #DFP | PGT(s) |
+-----+
| peruggia_1.1 | 3664 | 59 | 235 | 462 | 1.862 |
+-----+

awmar@LAPTOP-SJ374BMJ MINGW64 /c/xampp/htdocs/DeFP (main)
$

```

Figure A3. The final results of analyzing the Peruggia application.

## References

1. Deshlahre, R.; Tiwari, N. A Review on Benchmarking: Comparing the Static Analysis Tools (SATs) in Web Security. In *Social Networking and Computational Intelligence*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 327–337. [\[CrossRef\]](#)
2. Chang, J.; Gao, B.; Xiao, H.; Sun, J.; Cai, Y.; Yang, Z. sCompile: Critical path identification and analysis for smart contracts. In *International Conference on Formal Engineering Methods*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 286–304. [\[CrossRef\]](#)
3. Marashdih, A.W.; Zaaba, Z.F.; Almufti, S.M. The Problems and Challenges of Infeasible Paths in Static Analysis. *Int. J. Eng. Technol.* **2018**, *7*, 412–417.
4. Nazarahari, M.; Khanmirza, E.; Doostie, S. Multi-objective multi-robot path planning in continuous environment using an enhanced genetic algorithm. *Expert Syst. Appl.* **2019**, *115*, 106–120. [\[CrossRef\]](#)
5. Choma Neto, J. Automatic support for the identification of infeasible testing requirements. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, 18–22 July 2020; pp. 587–591. [\[CrossRef\]](#)
6. Sen, K.; Necula, G.; Gong, L.; Choi, W. MultiSE: Multi-path symbolic execution using value summaries. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 30 August–4 September 2015; pp. 842–853. [\[CrossRef\]](#)



7. Cadar, C.; Dunbar, D.; Engler, D.R. Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08), San Diego, CA, USA, 8–10 December 2008; pp. 209–224.
8. Havelund, K.; Pressburger, T. Model checking java programs using java pathfinder. *Int. J. Softw. Tools Technol. Transf.* **2000**, *2*, 366–381. [[CrossRef](#)]
9. Leißa, R.; Köster, M.; Hack, S. A graph-based higher-order intermediate representation. In Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), San Francisco, CA, USA, 7–11 February 2015; pp. 202–212. [[CrossRef](#)]
10. Lattner, C.A. LLVM: An Infrastructure for Multi-Stage Optimization. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, 2002.
11. Braun, M.; Buchwald, S.; Hack, S.; Leißa, R.; Mallon, C.; Zwinkau, A. Simple and efficient construction of static single assignment form. In *International Conference on Compiler Construction*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 102–122. [[CrossRef](#)]
12. Nguyen, D.A.; Huong, T.N.; Dinh, H.V.; Hung, P.N. Improvements of Directed Automated Random Testing in Test Data Generation for C++ Projects. *Int. J. Softw. Eng. Knowl. Eng.* **2019**, *29*, 1279–1312. [[CrossRef](#)]
13. Odeh, A.H. Analytical and Comparison Study of Main Web Programming Languages–ASP and PHP. *TEM J.* **2019**, *8*, 1517–1522. [[CrossRef](#)]
14. da Fonseca, J.C.C.M.; Vieira, M.P.A. A practical experience on the impact of plugins in web security. In Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems, Nara, Japan, 6–9 October 2014; pp. 21–30. [[CrossRef](#)]
15. Yang, J.; Lee, Y.; Fernandez, A.; Sanchez, J. Evaluating and Securing Text-Based Java Code through Static Code Analysis. *J. Cybersecur. Educ. Res. Pract.* **2020**, *2020*, 3.
16. Marashdih, A.W.; Zaaba, Z.F. Cross site scripting: Detection approaches in web application. *Int. J. Adv. Comput. Sci. Appl.* **2016**, *7*, 155–160.
17. Jiang, S.; Wang, H.; Zhang, Y.; Xue, M.; Qian, J.; Zhang, M. An Approach for Detecting Infeasible Paths Based on a SMT Solver. *IEEE Access* **2019**, *7*, 69058–69069. [[CrossRef](#)]
18. Aïssat, R.; Voisin, F.; Wolff, B. Infeasible Paths Elimination by Symbolic Execution Techniques. In *International Conference on Interactive Theorem Proving*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 36–51. [[CrossRef](#)]
19. Rocha, R.C.; Petoumenos, P.; Wang, Z.; Cole, M.; Leather, H. Effective function merging in the SSA form. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, London, UK, 15–20 June 2020; pp. 854–868. [[CrossRef](#)]
20. Léchenet, J.C.; Blazy, S.; Pichardie, D. A Fast Verified Liveness Analysis in SSA Form. In *International Joint Conference on Automated Reasoning*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 324–340. [[CrossRef](#)]
21. Quiroga, J.; Ortin, F. SSA transformations to facilitate type inference in dynamically typed code. *Comput. J.* **2017**, *60*, 1300–1315. [[CrossRef](#)]
22. Lin, Y. Symbolic Execution with Over-Approximation. Ph.D. Thesis, University of Melbourne, Melbourne, Australia, December 2017.
23. Godefroid, P.; Klarlund, N.; Sen, K. DART: Directed automated random testing. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL, USA, 12–15 June 2005; pp. 213–223.
24. Sen, K.; Marinov, D.; Agha, G. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Softw. Eng. Notes* **2005**, *30*, 263–272. [[CrossRef](#)]
25. Cadar, C.; Ganesh, V.; Pawlowski, P.M.; Dill, D.L.; Engler, D.R. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur. (TISSEC)* **2008**, *12*, 1–38. [[CrossRef](#)]
26. Williams, N.; Marre, B.; Mouy, P.; Roger, M. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *European Dependable Computing Conference*; Springer: Berlin/Heidelberg, Germany, 2005; pp. 281–292. [[CrossRef](#)]
27. Sen, K.; Kalasapur, S.; Brutch, T.; Gibbs, S. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 18–26 August 2013; pp. 488–498. [[CrossRef](#)]
28. Bucur, S.; Ureche, V.; Zamfir, C.; Candea, G. Parallel symbolic execution for automated real-world software testing. In Proceedings of the Sixth Conference on Computer Systems, Salzburg, Austria, 10–13 April 2011; pp. 183–198. [[CrossRef](#)]
29. Li, G.; Li, P.; Sawaya, G.; Gopalakrishnan, G.; Ghosh, I.; Rajan, S.P. GKLEE: Concolic verification and test generation for GPUs. In Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, New Orleans, LA, USA, 25–29 February 2012; pp. 215–224. [[CrossRef](#)]
30. Sasnauskas, R.; Landsiedel, O.; Alizai, M.H.; Weise, C.; Kowalewski, S.; Wehrle, K. KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks, Stockholm, Sweden, 12–16 April 2010; pp. 186–196. [[CrossRef](#)]
31. Li, G.; Ghosh, I.; Rajan, S.P. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In *International Conference on Computer Aided Verification*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 609–615. [[CrossRef](#)]
32. Cytron, R.; Ferrante, J.; Rosen, B.K.; Wegman, M.N.; Zadeck, F.K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **1991**, *13*, 451–490. [[CrossRef](#)]

33. Shafiei, N.; Breugel, F.V. Automatic handling of native methods in Java PathFinder. In Proceedings of the 2014 International SPIN Symposium on Model Checking of Software, San Jose, CA, USA, 21–23 July 2014; pp. 97–100. [\[CrossRef\]](#)
34. Nguyen, D.A.; Hung, P.N.; Nguyen, V.H. A method for automated unit testing of C programs. In Proceedings of the 2016 3rd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS), Hanoi, Vietnam, 14–16 September 2016; pp. 17–22. [\[CrossRef\]](#)
35. Nelson, G.; Oppen, D.C. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **1979**, *1*, 245–257. [\[CrossRef\]](#)
36. De Moura, L.; Bjørner, N. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 337–340. [\[CrossRef\]](#)
37. Gil Cepeda, J. Test Coverage of Systems with Continuous Dynamics. Master's Thesis, Chalmers University of Technology, Gothenburg, Sweden, 2017.
38. Popov, N. PHP-Parser. Available online: <https://github.com/nikic/PHP-Parser> (accessed on 29 August 2020).
39. Schardl, T.B.; Moses, W.S.; Leiserson, C.E. Tapir: Embedding recursive fork-join parallelism into LLVM's intermediate representation. *ACM Trans. Parallel Comput. (TOPC)* **2019**, *6*, 1–33. [\[CrossRef\]](#)
40. PHP. Superglobals. Available online: <https://www.php.net/manual/en/language.variables.superglobals.php> (accessed on 15 July 2020).
41. Zhang, X.; Hong, W.; Li, Y.; Sun, M. Reasoning about connectors using Coq and Z3. *Sci. Comput. Program.* **2019**, *170*, 27–44. [\[CrossRef\]](#)
42. Rapps, S.; Weyuker, E.J. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.* **1985**, *SE-11*, 367–375. [\[CrossRef\]](#)
43. Hermadi, I.; Lokan, C.; Sarker, R. Dynamic stopping criteria for search-based test data generation for path testing. *Inf. Softw. Technol.* **2014**, *56*, 395–407. [\[CrossRef\]](#)
44. Hermadi, I. Path Testing Using Genetic Algorithm. Ph.D. Thesis, University of New South Wales, Sydney, Australia, 2015.
45. Bueno, P.M.S.; Jino, M. Identification of potentially infeasible program paths by monitoring the search for test data. In Proceedings of the ASE 2000: Fifteenth IEEE International Conference on Automated Software Engineering, La Jolla, CA, USA, 16–19 July 2000; pp. 209–218. [\[CrossRef\]](#)
46. Jones, B.F.; Sthamer, H.H.; Eyres, D.E. Automatic structural testing using genetic algorithms. *Softw. Eng. J.* **1996**, *11*, 299–306. [\[CrossRef\]](#)
47. Bueno, P.M.S.; Jino, M. Automatic test data generation for program paths using genetic algorithms. *Int. J. Softw. Eng. Knowl. Eng.* **2002**, *12*, 691–709. [\[CrossRef\]](#)
48. Ahmed, M.A.; Hermadi, I. GA-based multiple paths test data generator. *Comput. Oper. Res.* **2008**, *35*, 3107–3124. [\[CrossRef\]](#)
49. Alba, E.; Chicano, F. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. *Comput. Oper. Res.* **2008**, *35*, 3161–3183. [\[CrossRef\]](#)
50. Blanco, R.; Tuya, J.; Adenso-Díaz, B. Automated test data generation using a scatter search approach. *Inf. Softw. Technol.* **2009**, *51*, 708–720. [\[CrossRef\]](#)
51. Sagarna, R.; Yao, X. Handling constraints for search based software test data generation. In Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, Lillehammer, Norway, 9–11 April 2008; pp. 232–240. [\[CrossRef\]](#)
52. Apiwattanapong, T.; Santelices, R.; Chittimalli, P.K.; Orso, A.; Harrold, M.J. Matrix: Maintenance-oriented testing requirements identifier and examiner. In Proceedings of the Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06), Windsor, UK, 4–6 September 2006; pp. 137–146. [\[CrossRef\]](#)
53. Wong, W.E.; Mathur, A.P. Reducing the cost of mutation testing: An empirical study. *J. Syst. Softw.* **1995**, *31*, 185–196. [\[CrossRef\]](#)
54. Papadakis, M.; Malevris, N. Automatic mutation test case generation via dynamic symbolic execution. In Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering, San Jose, CA, USA, 1–4 November 2010; pp. 121–130. [\[CrossRef\]](#)
55. PHP. Microtime. Available online: <https://www.php.net/manual/en/function.microtime.php> (accessed on 12 September 2020).
56. Roselli, S.F.; Bengtsson, K.; Åkesson, K. SMT solvers for job-shop scheduling problems: Models comparison and performance evaluation. In Proceedings of the 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE), Munich, Germany, 20–24 August 2018; pp. 547–552. [\[CrossRef\]](#)
57. Malyshev, N.; Dudina, I.; Kutz, D.; Novikov, A.; Vartanov, S. SMT Solvers in Application to Static and Dynamic Symbolic Execution: A Case Study. In Proceedings of the 2019 Ivannikov Ispras Open Conference (ISPRAS), Moscow, Russia, 5–6 December 2019; pp. 9–15. [\[CrossRef\]](#)
58. Weber, T.; Conchon, S.; Déharbe, D.; Heizmann, M.; Niemetz, A.; Reger, G. The SMT competition 2015–2018. *J. Satisf. Boolean Model. Comput.* **2019**, *11*, 221–259. [\[CrossRef\]](#)
59. Williams, B.; Tadlock, J.; Jacoby, J.J. *Professional WordPress Plugin Development*; John Wiley & Sons: Hoboken, NJ, USA, 2020; doi:10.1002/9781119666981. [\[CrossRef\]](#)
60. Thompson, S.; Dowrick, T.; Ahmad, M.; Xiao, G.; Koo, B.; Bonmati, E.; Kahl, K.; Clarkson, M.J. SciKit-Surgery: Compact libraries for surgical navigation. *Int. J. Comput. Assist. Radiol. Surg.* **2020**, *15*, 1075–1084. [\[CrossRef\]](#)

61. Liddell, C.; Kim, D. Analyzing the Adoption Rate of Local Variable Type Inference in Open-source Java 10 Projects. *J. Ark. Acad. Sci.* **2019**, *73*, 51–54.
62. Coelho, J.; Valente, M.T.; Silva, L.L.; Hora, A. Why we engage in FLOSS: Answers from core developers. In Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering, Gothenburg, Sweden, 27 May 2018; pp. 114–121. [[CrossRef](#)]
63. Derderian, K.; Hierons, R.M.; Harman, M.; Guo, Q. Estimating the feasibility of transition paths in extended finite state machines. *Autom. Softw. Eng.* **2010**, *17*, 33–56. [[CrossRef](#)]
64. Papadakis, M.; Malevris, N. A symbolic execution tool based on the elimination of infeasible paths. In Proceedings of the 2010 Fifth International Conference on Software Engineering Advances, Nice, France, 22–27 August 2010; pp. 435–440. [[CrossRef](#)]
65. Blackham, B.; Heiser, G. Sequoll: A framework for model checking binaries. In Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), Philadelphia, PA, USA, 9–11 April 2013; pp. 97–106. [[CrossRef](#)]
66. Huong, T.N.; Tran, H.V.; Hung, P.N. Generate Test Data from C/C++ Source Code using Weighted CFG and Boundary Values. In Proceedings of the 2020 12th International Conference on Knowledge and Systems Engineering (KSE), Can Tho, Vietnam, 12–14 November 2020; pp. 97–102. [[CrossRef](#)]
67. Wang, Z.; Han, W.; Lu, Y.; Xue, J. A Malware Classification Method Based on the Capsule Network. In *International Conference on Machine Learning for Cyber Security*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 35–49. [[CrossRef](#)]