International Journal of Engineering & Technology, 7 (4.19) (2018) 412-417

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper



The Problems and Challenges of Infeasible Paths in Static Analysis

Abdalla Wasef Marashdih[1], Zarul Fitri Zaaba [1], Saman M. Almufti [2]

[1]School of Computer Sciences, Universiti Sains Malaysia, 11800 Minden, Pulau Pinang, Malaysia
[2]College of Computer Science and Information Technology, Department of Computer Science, Nawroz University
**Corresponding Author Email: a.w.marashdih@gmail.com*

Abstract

Static analysis is valuable because it imparts the ability to examine all program paths. However, many of these paths are classified as infeasible paths, which signify that these paths will fail to execute. In static analysis, these paths will lead to results that are high false positive. Because static analysis has a vital part in the detection of vulnerabilities and threats in the software as well as in program analysis, improving static analysis is necessary to obtain accurate results and lessen the occurrence of false positive results. Being able to detect infeasible paths is useful in the improvement and development of the results of static analysis. However, the process that is used to identify these infeasible paths is not simple, especially because numerous tools and methods still do not have the efficiency in detecting these kinds of paths within the static analysis. This paper will review the infeasible paths problem in the static analysis, the new methods of solving this problem, and the reassessment of this vital issue in software testing. This paper will also discuss the importance of exposing and getting rid of these paths.

Keywords*: *Infeasible Paths; Path Testing; Static Analysis; Software Testing, Security; Vulnerabilities.

1. Introduction

TATIC analysis is seen as one of the most essential methods for source code analysis because it can detect security problems and vulnerabilities even before the program is executed for the first time [1]. Static analysis encompasses the entire source code, making it the most effective tool for detecting threats and security vulnerabilities in the source code [2]. However, false positive results are often obtained in static analysis, because static analysis sometimes analyses paths that are considered infeasible paths, or the paths that fail to execute. This is why the false positive results problem is a major issue in static analysis that needs to be examined and addressed. Eliminating infeasible paths from the paths that will undergo static analysis will lower the rate of false positives in the results [3].

According to Hedley and Hennell [4], 12.5% of the whole paths can be classified as infeasible paths, signifying that they will not execute regardless of the type of input data. Eliminating these paths will improve static analysis since its results will be improved by the reduction of the false positive rate, which constitutes a major static analysis limitation.

Numerous methods and tools have been utilised to identify infeasible paths, but these tools and methods still do not have enough efficiency for the detection of these paths. Symbolic execution is one of the techniques for the detection of infeasible paths within the source code [5,6]. However, the utilisation of symbolic execution raises the cost for sharper analytical results. Furthermore, it can detect a small amount of infeasible paths since it lessens symbolic evaluation in function calls and arrays.

Other researchers utilised genetic algorithm to generate only feasible paths and avoid infeasible paths [7,8,9]. They were able to achieve promising results since the infeasible paths were avoided. However, certain instruments within their genetic algorithm function manually, which uses up a significant amount of time and manpower. Their approach is therefore not appropriate for the large and complex program. One can also utilise dynamic test data generation algorithms to detect infeasible paths. It does this by monitoring the program's execution [10,11]. However, symbolic execution is often utilised for test data generation making it almost as costly as symbolic execution.

It should be noted that identifying a solution to detect all the infeasible paths within the source code is difficult. This paper discusses static analysis and its main issue, which is infeasible paths. It will also talk about the proposed approaches for discovering the infeasible paths.

The succeeding parts of the paper will be organised in the following manner. Section II focused on the related approaches for detecting infeasible paths. Section III and section IV will discuss the concept and the strategy behind the static analysis as well as the infeasible paths. Section V will talk about the recent methods that have been utilised for detecting the infeasible paths and discusses the advantages that come with detecting infeasible paths. Conclusion and future works are discussed in section VI.

2. Static Analysis

Static analysis is used in the analysis of source code for programs and for finding and identifying weaknesses within the source code. Static analysis has the capacity to view all the possible program paths found within the application, which helps in determining the coverage for all the paths of the program [12]. Static analysis has been widely used to analyse programs and applications so

Copyright © 2018 Authors. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

that gaps and the weaknesses can be found. The data analysis process in the static analysis has its basis on the control flow graph and the code's data flow analysis. This process does not need any real implementation of the program.

The standard way of representing the data program's flow is the Control Flow Graph (CFG) [13]. Every sentence is represented using a node that is linked together by edges so that the program's data flow can be represented. A starting node starts the process. The flow then starts and each node (statement) is represented until the program ends. Every path that goes from the starting node all the way to the end node is taken as the data's logic path in the program. In order for the path to be implemented in the CFG, the inputs have to satisfy the conditions that all the branches of the path impose. Figure 1 illustrates an example of the PHP source code and the CFG for that example.

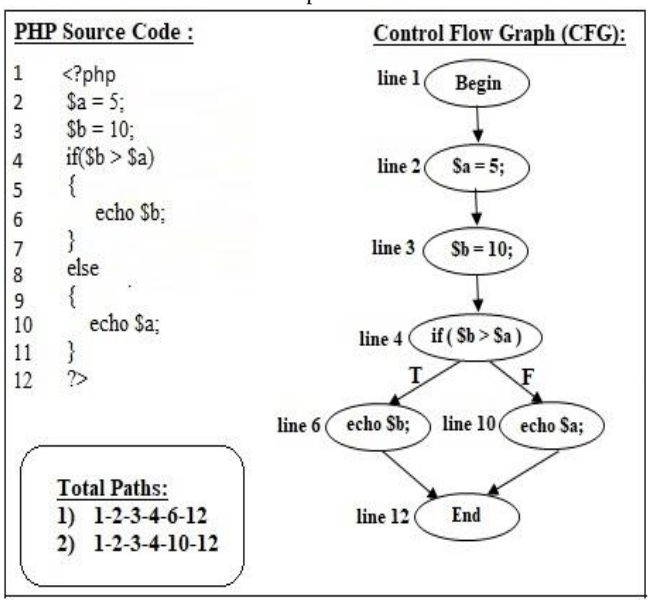


Fig. 1: Example of PHP Source Code and CFG

The PHP begins by giving a definition to the two variables ($a = 5 and $b = 10). The condition given at line 4 then makes a comparison between the two variables ($b > $a). The statement (echo $b) found at line 6 will be executed if the condition is TRUE. Otherwise, it would execute the statement (echo $a) at line 10.

Because static analysis tools are able to detect errors within the paths that were made from the control flow graph, there is a great chance for these paths to not be executable (infeasible paths). These lower the accuracy of the results, especially the ones that are assumed to have errors. These paths are often infeasible, which means that the tool does not need to consider them or allot time for their analysis. The following section will talk about the infeasible paths within the static analysis. It will also discuss how it works.

3. Infeasible Paths

One can define the infeasible path as any path within the CFG that is not executable under any input values or any test cases [13,16]. Figure 2 illustrates an example of the PHP source code with its control flow graph and shows how some paths end up being classified as infeasible paths.

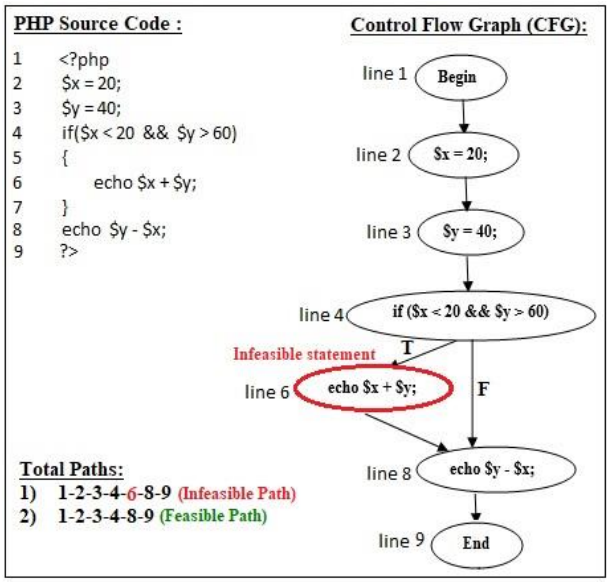


Fig. 2: PHP Source Code and the Infeasible Paths in CFG

Figure 2 shows that the static analysis begins by analysing the PHP source code and creating the source code's control flow graph so that the program's data flow can be determined. The PHP source code begins with the initialisation of the two variables found in line 2 and 3 ($x = 20 and $y = 40). Then, a condition specified in line 4 will check if ($x < 20) and ($y > 60). The statement in line 6 (echo $x + $y) is implemented if the condition is true. It then ends at line 8 (echo $y - $x). However, if the statement presented in line 6 (echo $x + $y) is an infeasible statement, there will be no probability for the given condition to be true at all. Thus, every path that has the line 6 implementation will be considered an infeasible path.

Infeasible paths can exist because of several reasons [13], one of which is the existence of dead code, which means that it is not possible to implement a certain sentence within the code. Consequently, they result into unimplementable paths because accessibility for this data does not exist. Another cause of infeasible paths is the conflicting clauses that are contained within certain paths, such as: (A <= 10 && B > 20 || C == 40). Infeasible paths may also be present as a result of the existence of correlated conditional statements in terms of a given variable (which is considered as one of the main reasons for the absence of access in the programs [14, 15]).

To enhance the results of the static analysis, one should remove infeasible paths from the entire path of the control flow graph [3, 13, 16, 17]. One can define the infeasible path as any path that is not executable under any test cases [13]. Conversely, Ball, T. and Balakrishnan, et al. [16,17] stated that developers need to distinguish these infeasible paths from the other paths of the entire control flow graph. The following section will provide a description of the detection of infeasible paths as well as the methods that were used to identify these infeasible paths.

4. Detection of Infeasible Paths

It is very important to detect infeasible paths. Furthermore, instead of trying to test them, discovering these paths will help conserve time and resources [13]. The presence of these infeasible paths influences numerous fields of software engineering. Detecting these paths will improve the analysis and detection process of security vulnerabilities [3,18], help conserve time, and improve accuracy.

It also helps in checking the web application [19,20,21,22,23] as

well as the database applications design [14,24], which are both valuable in the detection of these infeasible paths.
Complex data structures and dependencies need to be dealt with when detecting infeasible paths. Numerous tools and methods have been utilised for the detection of infeasible paths. However, these methods and tools still do not have enough efficiency to detect these paths. Table 1 shows the approaches used to detect infeasible paths.

Table 1: Approaches Employed for Detecting Infeasible Paths

Author	Approach Description	Results	Supported Language
Ngo and Tan [25] (2007)	An innovative method for determining infeasible paths in four common code patterns	Detects 82.3% of all infeasible paths	XML
Ngo and Tan [14] (2008)	A heuristics-based methodology for infeasible path detection and dynamic test data generation	Detects 96.02% of all infeasible paths	JAVA
Papadakis and Malevris[5] (2010)	An automated symbolic implementation tool	Detects 93% of 50 program paths per branch	Delphi, C/C++
Gong and Yao [15] (2010)	Automatic static analysis and dynamic techniques; identifies the brunch correlations for ascertaining infeasible paths	Detects 99.81% of all infeasible paths	C
Yano, et al., [26] (2011)	The MOST method utilises a multi-objective evolutionary algorithm and an objective function	Can effectively ensure path feasibility	JAVA
Wong, et al., [28] (2013)	Modified breadth first search with conflict checker	Promising outcomes for determining path feasibility	C++
Jayaraman and Tragoudas [29] (2013)	Control as well as data dependency are taken into account for determining the infeasible paths.	Not effective in determining infeasible paths	C++
Hermadi, et al., [30] (2014)	Genetic algorithm with decision rules	The recommended methodology appears to be helpful with few missed feasible paths	C
Ruiz, and Cassé [31] (2015)	Depiction of program states as labelled sets of predicates with Satisfiability Modulo Theories (SMT) solver	Not effective in covering the whole infeasible paths in the program	C++
Delahaye, et al., [6] (2015)	Generalise infeasible paths from the uncovering of a single infeasible path with DSE-based automated test input generation	The method can save substantial computation time during test generation	C
Ruiz, et al., [32] (2017)	An infeasible path lookup analysis which benefits from being composed and split, and the SESE regions comprising the bodies of subroutines and loops	Diminution of the WCET by more than 10%; many kinds of infeasible paths remain untraceable	C
Marashdih, et al., [18] (2017)	Manual removal of infeasible paths in control flow graph	More precise outcomes in software testing; eliminating infeasible paths should be made automatic	PHP

Table 1 shows that numerous research works are still focused on removing these infeasible paths from the control flow graph. Despite this, these approaches still do not have adequate efficiency to detect these kinds of paths. Conversely, the removal of infeasible paths is seen as a vital stage in improving the result of static analysis. The approach developed by Marashdih et al. [18] serves as a real example of how software testing can be improved once the infeasible paths are removed. They were able to achieve accurate results without having any incidence of false positives. They stated that they were able to produce these results mainly because the infeasible paths have been removed.

Majority works for eliminating infeasible paths emphasise the C, C++, Java programming languages. However, the removal of infeasible paths is vital for the other programming languages like ASP.Net, PHP and Python. More importantly, programming languages that are utilised for building applications make use of static analysis for the detection of any security vulnerability within these applications. The following section will present more details regarding the approaches that were utilised to detect infeasible paths.

5. Related Work

A number of works have been previously published regarding the topic of detecting infeasible paths. Ngo and Tan [25] formulated a method to detect infeasible paths. Binomial tests were performed

and these tests gave strong statistical proof that supported the validity of the empirical properties. Based on their experimental results on XML, they discovered despite certain limitations in the present prototype tool that the proposed approach was able to detect 82.3% of all the infeasible paths accurately. In 2008, Ngo and Tan suggested a heuristics-based approach that can be used to detect any infeasible path for the generation of dynamic test data. Their experiments, which were performed on Java source code, revealed that the proposed approach was able to detect majority of the infeasible paths efficiently, with an average precision value of 96.02% and a recall value that is at 100% for all the cases.

Papadakis and Malevvris [5] formulated an automated symbolic execution tool that can be used for the detection of infeasible paths. The tool makes use of an efficient path heuristic, which is then integrated with random testing so that test cases can be produced. The tool is able to efficiently handle the explosion of the path and the constraint of solving problems. This is attained by targeting specific paths that are likely to be feasible and then utilising a linear programming approach to determine their feasibility. The preliminary results they obtained have shown great promise as they revealed that one can obtain high coverage for a limited amount of time-effort. Their results revealed that the tool they developed is able to detect 93% of the 50 program paths for every brunch.

Gong and Yao [15] formulated a tool for automatically identifying the branch correlations of various conditional statements, which in turn helps detect infeasible paths. In this technique, the advantages of dynamic techniques and static analysis are combined. This method also identifies the branch correlations of various conditional statements by using the maximum likelihood estimation. First, it provides some theorems that can be used to identify branch correlations that are based on the probabilities that the conditional distribution will correspond to the outcomes of different branches (i.e. true or false); then, it uses maximum likelihood estimation to obtain values for these probabilities; lastly, it detects infeasible paths based on branch correlations. The proposed method was applied in some typical C programs, with the results revealing that the proposed method is capable of accurately detecting infeasible paths. The achievement is able to offer an automatic and effective method of infeasible path detection, which is important in enhancing the efficiency of software testing.

To avoid the generation of an infeasible path, Yano, et al. [26] proposed the MOST approach. This approach is a search-based testing technique for the generation of a test case from Extended Finite State Machines (EFSM). MOST makes use of a multi-objective evolutionary algorithm so that the generation of test cases will be able to cover a given transition (test purpose). It can then find more than one successful path to cover the proposed test. To serve as a guide to the search for a test purpose, an objective function was proposed. This function makes use of information that has been obtained from a dependence analysis of the model. This makes sure that the solution formulated is able to cover most of the transitions that the test purpose is dependent on. They take both control and data dependence analysis into consideration. The results obtained from MOST were then compared to the results obtained from another search based testing approach for EFSM [27]. The MOST results obtained generally similar or even better results.

Wong, et al. [28] suggested a method that utilises the modified breadth first search with conflict checker so that a set of minimum Feasible Transition Path (FTP) can be generated for each transition. They developed an EFSM executable model for algorithm modelling, algorithm verification, and performance assessment. Experimental results that were performed on two EFSM models revealed that their proposed approach is capable of generating feasible transition paths that have at least 18% reduction in path length.

Jayaraman and Tragoudas [29] proposed a new algorithm to recognise unfeasible pathing in the behavioural code. The initial step of the proposed strategy partitions behavioural codes into segments. For every segment, it implicitly stores every possible path. Similarly, it stores input assignment sets that derive from certain statements in the segments of code. The technique requires advanced data structures for storing possible paths and necessary functions. Experimental findings have established the scalability of this approach.

Hermadi, et al. [30] presented and evaluated a technique for determining when further searches for test data that covers exposed target paths are no longer worthwhile. The main parameters as well as uses in various decision rulesets for advance terminations of searches of the approach are outlined. The key advantage of their recommended strategy is that arbitrary parameters (the number of generations to be searched) are substituted with a technique that accords with search history, while the researcher's decision on when to halt tests can accord instead with the likelihood those additional tests will not contain further pathing, to include the stability of the probability. A twenty-one test program set from SBSE path-testing studies was utilised to assess the technique. In comparison to searches comprising standard number of generations, 30% to 75% of the computational burden on average was evaded in program tests with unfeasible pathing, with no possible pathing missed as a result of early terminations. Additional computation including unfeasible pathing was insignificant; the approach is efficient and successful. It circumvents the requirement for specifying limits to the number of search generations and can assist in overcoming difficulties due to unfeasible pathing in search-based generation of data in tests for paths.

Ruiz, and Cassé [31] presented a new strategy for discovering unfeasible pathing in binary programs. Their approach consists of static analyses of (a) a CFG with blocks composed of machine instructions (abstracted via semantic instruction sets) and (b) of a programmatic data state symbolised by register and memory predicates. SMT unsatisfiability in predicates enables identification of unfeasible pathing, resulting in a listing of edges from the program CFG that are prohibited on possible execution paths. Such information is normally utilised to enhance the accuracy of WCET computations. Nevertheless, certain unfeasible paths will not be discovered due to (a) overly coarse states that join operators and (b) exploding time calculations.

Delahaye, et al. [6] explain a new approach to generalised unfeasible pathing from detections of single unfeasible paths, as well as a method for exploiting this unfeasible-path generalisation technique, for DSE-based automated input generation in testing. The technique comprises three steps. Firstly, it derives explanations of infeasibility; Secondly, it determines dependencies in the data that associate with given explanations; Thirdly, it constructs automations that generalise fed unfeasible paths, allowing users to detect early on other unfeasible paths that share similar explanations. The recommended approach has been applied to common DSE-based input generation testing processes. Generic processes were used so as to evaluate the technique apart from particular constraint solvers or descriptive methods. Experimental findings obtained through this method demonstrated that this unfeasible path generalisation strategy compares favourably with that of exhaustive unfeasible path detections, in that the method can accelerate DSE when generating test inputs.

Ruiz, et al. [32] presented a reasonably scalable strategy for unfeasible path detection in binary programs that handle loop and function call processes, using a method that stores sufficient information about the programmatic state for the detection of significant unfeasible pathing within loops. The analytical framework

relies on composed representations of the states of programs. It can be applied to localised code segments, including subroutine and loop parts. Analysis can also assist in outputting unfeasible pathing of the greatest possible scope in general, therefore amplifying the effect on WCET estimations. Experimental tests using Malardalen benchmarks [33] showed conclusive outcomes, such as a decrease of over 10% in the burden of WCET estimations, as conducted in three large benchmarks. Nevertheless, various types of unfeasible pathing remain indiscernible to their analytical approach. This limitation includes inter-loop conflicts, as in the instance of paths that are not resolvable within two successive iterations.

Marashdih, et al. [18] proposed a strategy for identifying cross site scripting (XSS) in PHP in accordance with genetic algorithms and static analyses, and another strategy to remove the detected XSS vulnerability from the source code [34,35]. The technique improves on the previous strategy of Ahmad and Ali [36] by eliminating unfeasible pathing from control flow graphs. This assists in enhancing findings and leads to more accurate outcomes than that of Ahmad and Ali [36]. But as unfeasible pathing was eliminated manually, the strategy applies only to smaller programs. Some method for automatically removing every unfeasible path must be determined for their strategy to apply to larger programs as well.

6. Conclusion

Static analysis is termed as one of the most vital methods for scrutinising the source code. It has the capability to view all likely program paths in the application; this aids in determining the coverage for the entire paths of the program. Because static analysis tools are able to detect errors within the paths that were made from the control flow graph, there is a great chance for these paths to be not executable (infeasible paths), which make the outcomes not precise, particularly those which are presumed to have errors. These paths are often infeasible, which means that the tool does not need to consider them or allot time for their analysis. Thus, we have elucidated the infeasible path problem in static analysis accompanied by examples. Furthermore, we outline the detection and the methodologies deployed for determining infeasible paths. It is noteworthy that all current approaches cannot determine most of the infeasible paths effectively. The majority of the approaches on eliminating infeasible paths emphasise on C, C++, Java programming languages. However, eliminating infeasible paths is vital for all programming languages like Python, PHP, and ASP.net.

Acknowledgement

"This research is fully supported by University Research Grant from Sultan Idris Education University under the grant number of 2018-0134-109-01."

References

[1] Da Fonseca, J. C. C. Martin, and M. P. A. Vieira, "A practical experience on the impact of plugins in web security," in 2014 IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS), pp. 21-30.

[2] A., Avizienis,J. C.Laprie, B.Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing,"IEEE transactions on dependable and secure computing, vol. 1, no. 1, pp. 11-33, Jan. 2004.

[3] A. W.Marashdih, and Z. F.Zaaba,"Cross Site Scripting: Detection Approaches in Web Application,"(IJACSA) International Journal of Advanced Computer Science and Applications, vol. 7, no. 10, Oct. 2016.

[4] D. Hedley, and M. A. Hennell, "The causes and effects of infeasible paths in computer programs,"in 1985 Proceedings of the 8th international conference on Software engineering, pp. 259-266.

[5] M., Papadakis, and N. Malevris, "A symbolic execution tool based on the elimination of infeasible paths," in 2010 Fifth International Conference on Software Engineering Advances (ICSEA), pp. 435-440.

[6] M., Delahaye, B. Botella, and A. Gotlieb, "Infeasible pathgeneralization in dynamic symbolic execution,"Information and Software Technology, vol. 58, pp. 403-418, Feb. 2015.

[7] A. S.Ghiduk, "Automatic generation of basis test paths using variable length genetic algorithm," Information Processing Letters, vol. 114, no. 6, pp. 304-316, Jun. 2014.

[8] M. A., Ahmed, and I.Hermadi, "GA-based multiple paths test data generator,"Computers & Operations Research, vol. 35, no. 10, pp. 3107-3124, Oct. 2008.

[9] D.Gong, W.Zhang, and X. Yao, "Evolutionary generation of test data for many paths coverage based on grouping,"Journal of Systems and Software, vol. 84, no.12, pp. 2222-2233, Dec. 2011.

[10] P. M. S.Bueno, and M. Jino, "Identification of potentially infeasible program paths by monitoring the search for test data," in2000 Proceedings Fifteenth IEEE International Conference on Automated Software Engineering, ASE, pp. 209-218, Sep. 2011.

[11] N.Gupta, A. P.Mathur, and M. L.Soffa,"Generating test data for branch coverage,"in 2000 proceedings Automated Software Engineering, ASE, pp. 219-227, Sep. 2000.

[12] V.Prokhorenko, K. K. R.Choo, and H. Ashman, "Web application protection techniques: A taxonomy,"Journal of Network and Computer Applications, vol. 60, pp. 95-112, Jan. 2016.

[13] B.Barhoush, and I. Alsmadi, "Infeasible Paths Detection Using Static Analysis,"The Research Bulletin of Jordan ACM, vol. 2, no. 3, pp. 120-126, 2013.

[14] M. N.Ngo, and H. B. K.Tan, "Heuristics-based infeasible path detection for dynamic test data generation,"Information and Software Technology, vol. 50, no. 7-8, pp. 641-655, Jun. 2008.

[15] D. Gong, and X.Yao,"Automatic detection of infeasible paths in software testing," IET software, vol. 4, no. 5, pp. 361-370, Oct. 2010.

[16] G.Balakrishnan, S.Sankaranarayanan, F.Ivančić, O.Wei, and A.Gupta, "SLR: Path-sensitive analysis through infeasible-path detection and syntactic language refinement,"in 2008 International Static Analysis Symposium, pp. 238-254.

[17] T. Ball, "Paths between Imperative and Functional Programming,"ACM SIGPLAN Notices, vol. 34, no. 2, pp. 21-25, Feb. 1999.

[18] A. W.Marashdih, Z. F.Zaaba, and H. K.Omer, "Web Security: Detection of Cross Site Scripting in PHP Web Application using Genetic Algorithm,"International Journal of Advanced Computer Science and Applications (IJACSA), vol. 8, no. 5, May 2017.

[19] S.Ding, and H. B. K.Tan, "Detection of Infeasible Paths: Approaches and Challenges,"in International Conference on Evaluation of Novel Approaches to Software Engineering, Jun 2012, pp. 64-78.

[20] H.Liu, and H. B. K. Tan, "Covering code behavior on input validation in functional testing," Information and Software Technology, vol. 51, no. 2, pp. 546-553, Feb. 2009.

[21] H.Liu, and H. B. K. Tan, "Testing input validation in Web applications through automated model recovery,"Journal of Systems and Software, vol. 81, no. 2, pp. 222-233, Feb. 2008.

[22] H.Liu, and H. B. K. Tan, "An approach for the maintenance of input validation,"Information and Software Technology, vol. 50, no. 5, pp. 449-461, Apr. 2008.

[23] H.Liu, and H. B. K. Tan, "An approach to aid the understanding and maintenance of input validation,"in 2006 22nd IEEE International Conference on Software Maintenance, ICSM'06, pp. 370-379.

[24] M. N.Ngo, and H. B. K. Tan, "Applying static analysis for automated extraction of database interactions in web applications,"Information and software technology, vol. 50, no. 3, pp. 160-175, Feb. 2008.

[25] M. N.Ngo, and H. B. K.Tan,"Detecting large number of infeasible paths through recognizing their patterns,"in 2007 Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 215-224.

[26] T.Yano, E.Martins, and F. L.de Sousa, "MOST: a multi-objective search-based testing from EFSM," in 2011 IEEE Fourth Interna-

tional Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 164-173.
[27] A. S.Kalaji, R. M.Hierons, and S.Swift, "Generating feasible transition paths for testing from an extended finite state machine (EFSM)," in 2009 International Conference on Software Testing Verification and Validation, ICST'09,pp. 230-239.
[28] S.Wong, C. Y.Ooi, Y. W.Hau, M. N.Marsono, and N.Shaikh-Husin, "Feasible transition path generation for EFSM-based system testing," in 2013 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1724-1727.
[29] D.Jayaraman, and S.Tragoudas,"Performance validation through implicit removal of infeasible paths of the behavioral description," in 2013 14th International Symposium on Quality Electronic Design (ISQED), pp. 552-557.
[30] I.Hermadi, C.Lokan, and R.Sarker,"Dynamic stopping criteria for search-based test data generation for path testing,"Information and Software Technology, vol. 56, no. 4, pp. 395-407, Apr. 2014.
[31] J.Ruiz, and H.Cassé, "Using smt solving for the lookup of infeasible paths in binary programs," in OASIcs-OpenAccess Series in Informatics, vol. 47, 2015.
[32] J.Ruiz, H.Cassé, and M. de Michiel, "Working Around Loops for Infeasible Path Detection in Binary Programs," in 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 1-10.
[33] J.Gustafsson, A.Betts, A.Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks: Past, present and future," In OASIcs-OpenAccess Series in Informatics, vol. 15, 2010.
[34] A. W. Marashdih, and Z. F. Zaaba, "Cross Site Scripting: Removing Approaches in Web Application,"Procedia Computer Science, vol. 124, pp. 647-655, Dec. 2017.
[35] A. W. Marashdih, and Z. F. Zaaba, "Detection and Removing Cross Site Scripting Vulnerability in PHP Web Application," in 2017 International Conference on Promising Electronic Technologies (ICPET), pp. 26-31.
[36] M. A.Ahmed, and F. Ali, "Multiple-path testing for cross site scripting using genetic algorithms," Journal of Systems Architecture,vol. 64, pp. 50-62, Mar. 2016.