Detection Execution of Path through Dynamic Analysis in Black-Box Testing Environments

Abstract:

Path coverage is the technique of calculating the percentage of execution pathways that are taken by a given set of inputs during run-time in a software. In order to evaluate the stability, security, and Software testing is very directly related to the functioning of an application. When the source code is unavailable and path coverage must be performed using only the software's binary code, the issue becomes more difficult. Path coverage needs knowledge of the software's source code (white-box testing), especially the software's various execution pathways. This could happen if the program is a product, a legacy system, or if the source code is not accessible (as with contractual software or permission-free software). This work explores the use of execution fingerprints, a frequency representation of the executed assembly instructions in a software, for black-box route identification and discovery. The inputs that were used to exercise various parts of the code may be determined using execution fingerprints, which reveals the execution routes. A complete black-box testing environment may be utilized to discern between distinct software execution routes using clustering execution fingerprints, according to experimental results.

Introduction:

The investigation of a software's (during run-time) execution routes in relation to a specified test set is known as path coverage [1].

This study is valuable because it can provide light on the software's complexity, particularly when combined with a cutting-edge cyclomatic complexity metric that counts the number of linearly independent pathways that are run in software[2.]

White-box testing, which allows for the viewing of the source code, is often used to obtain path coverage.[3]

Source code might not constantly be accessible, though. For instance, if software is a product, its creators may restrict access to its source code or may be subject to copyright restrictions that limit the source code's exposure. Similar to how various systems are passed down over time and source code is not properly maintained in these systems, legacy software seldom makes its source code accessible to users. Lastly, a software's source code can simply not be available because it has been deleted, lost, or stolen, or because the user doesn't have the right access rights. Black-box testing has led to the development of tools for dynamic analysis, such decision table testing and equivalence partitioning. [4]

However, no research has been done on black-box techniques that can detect and find execution paths without any visibility into or speculation about the source code.

The unique technique for black-box testing described in this research is based on the execution of assembly instructions given various software inputs. An execution fingerprint, which is a brief, recognizable, and instructive depiction of the execution route, is created during program execution using the frequency of the executed assembly instructions. As a consequence, in a black-box testing environment, execution fingerprints may be used to recognize and learn about several execution routes given a group of software inputs. These execution fingerprints can successfully expose and discriminate execution patterns through generated clusters, according to preliminary testing employing clustering algorithms.

Related Work:

Path coverage tools have been created, as previously mentioned, to work with software in primarily white-box testing suites.[5][6]

There have been a few instances of path coverage attempts in environments that are nearly black boxes.[7]

In a black-box scenario, Tahat et al. investigated test set generation by path coverage. The strategy, nevertheless, is reliant on the Specification and Descriptive Language (SDL). Jiang, et al. investigated using existing flow charts (created from source code) to map out the executable's route coverage. Both of these methods go against the IEEE's definition of "black-box testing" since they need a great deal of knowledge about the design and workings of the program being tested.

Methodology:

Black-box testing typically entails observing the results produced when a software program is given particular inputs. The outputs of the software include both expected results (results for the intended task) and unexpected results, like a core-dump. Even though this information might further reveal the operation (via execution pathways) and general complexity of the software, the executed assembly instructions are rarely examined during the testing process. This makes sense considering the sheer volume of machine instructions required for even the simplest jobs.

In order to estimate the execution pathways followed and total program complexity in a black-box testing environment, this article employs assembly instruction fingerprints. The software instructions that were actually executed given a set of inputs are represented as a normalized frequency in an execution fingerprint. The number of times a certain instruction is carried out during program execution is noted.

The instruction counts are then normalized depending on the total number of instructions executed after the execution is complete. To give a more broad idea of an execution route, normalization is done. Due to repetition structures, for instance, inputs that follow the identical execution route may differ in the precise number of instructions carried out. In many cases, normalization can reduce these discrepancies by relying on percentages rather than absolute values.

Although an execution fingerprint can offer a very compact representation of an execution route, the representation can be strengthened by including or excluding code from external libraries. System input/output or memory library code management, which might make up a sizable amount of the instructions carried out. By looking at the memory location of a certain command, these two distinct execution types may be identified. External instructions may obfuscate the route representation that the execution fingerprint aims to offer, depending on the ratio of internal to external instructions utilized in the software. In order to offer a better route representation, it may be desirable to filter-out instructions linked to external libraries depending on the software.

Moreover, some assembly language opcodes can be translated to specific fields.

The technique that follows allows for the observation of program complexity and the number of pathways followed using execution fingerprints.

Afterwards, fingerprints may be made for each input using a set of software inputs (those used for black-box testing). These printouts can then be compared (e.g. clustered) to ascertain the quantity of software execution pathways taken when the set of inputs is applied.

If just one execution path is chosen, black-box testing may be judged insufficient; nevertheless, if several execution pathways are used, software complexity rises. Due to the limitations of black-box testing, sets of execution pathways followed during run-time are represented by these created fingerprints. This article describes how to group these groups of execution pathways depending.

Result:

The use of execution fingerprints for identifying execution paths is briefly discussed in this section. The range of effectiveness of created fingerprints was examined using a variety of sub-experiments. These smaller studies involved altering the filter. During assembling the software under test, assembly instructions are logged, and the optimization level is changed. Because they directly impact the quantity of assembly instructions utilized in the execution fingerprinting, these two characteristics were put to the test. By examining the memory location linked to the assembly instruction, it was possible to determine if the executed assembly instruction was called from an external C library method.

Software (created for this project) that ran the program under test as a Unix child process was used to create fingerprints. Each time a youngster follows a parent's instructions, the parent the procedure would note the frequency of the instruction. The normalized execution fingerprint was stored as a file when the execution was finished. The test program was created in C for these studies and needed a command line parameter (input) to function.

One of the four potential execution pathways was chosen based on the input given. Each execution route was made up of a sequence of function calls, each of which would read a file, multiply matrices, run Dijkstra's Algorithm on a directed graph, or do a binary search on an array of provided data (information for the operations). The program (to be tested) was given inputs during experiments in order for one of the four possible execution paths to be chosen.

After then, the fingerprints were grouped to see if the groups made sense, created do in fact reflect the four potential directions. ARI and silhouette ratings were used to evaluate how well the clustering performed.

By clustering using basic testing software, all four execution pathways were quickly identified during experimentation.

To evaluate the effectiveness of the execution fingerprints, several distinct pieces of software were created. Filtering away library-related assembly instructions while evaluating the various sub-experiments barely had an impact on the system's overall speed since it substantially reduced the amount of data that could be used to create execution fingerprints. Contrarily, full optimization compiled software outperformed zero optimization compiled software by a little margin, even though more optimization during compilation means fewer assembly instructions would be executed at runtime.

Although the tool's overall speed was impacted by these various setups, all execution pathways were recognized and found with a 100% success rate. The silhouette scores accounted for the majority of the performance difference.

Future Work:

To aid with larger testing and studies, future work will involve adding opcode subsequences to normalized fingerprints. Last but not least, further research will focus on cyclomatic complexity and bolster the case for it as a useful indicator of software complexity.

The main criticism of cyclomatic complexity has been its failure to accurately breakdown software into appropriate modules for accurate and comprehensive comparison.[8]

Future study will shed light on whether the decomposition of software into normalized execution fingerprints might deliver stable local complexity levels. This work proposes a unique dynamic analysis approach.

Conclusion:

In this study, a novel approach to dynamic analysis is used to identify and track software execution routes in a closed-loop testing environment. Clustering receives normalized fingerprints made up of completed assembly instructions, which then detect and

differentiating the run-time execution routes. It has not yet been possible to achieve path identification at this degree of black-box testing, but the findings of this experiment indicate that it will be possible in the future. Regression test set generation, operating system scheduling, parallel computing scheduling, and code review for software complexity can all benefit from this kind of analysis. It is possible to gain important insight into the system's run-time by identifying and classifying a software's execution paths, which increases visibility into a hidden black-box testing environment.

References:

1. B. Hom’s Fundamentals of software testing, 2012

2. Ebert, Christof, et al. "Cyclomatic complexity." IEEE software 33.6 (2016): 27-29.

3. Myers, Glenford J., Corey Sandler, and Tom Badgett. The art of software testing. John Wiley & Sons, 2011.

4. Khan, Mohd. "Different approaches to black box testing technique for finding errors." International Journal of Software Engineering & Applications (IJSEA) 2.4 (2011).

5. Gotlieb, Arnaud, and Matthieu Petit. "Path-oriented random testing." Proceedings of the 1st international workshop on Random testing. 2006.

6. Jian hua Sun and Shu juan Jiang. 2010. An approach to automatic generating test data

for multi-path coverage by genetic algorithm. In 2010 Sixth International Conference

on Natural Computation, Vol. 3. 1533–1536. https://doi.org/10.1109/ICNC.2010.

5583778

7. Jiang, Shujuan, Yanmei Zhang, and Dandan Yi. "Test data generation approach for basis path coverage." ACM SIGSOFT Software Engineering Notes 37.3 (2012): 1-7.

8. Shepperd, Martin. "A critique of cyclomatic complexity as a software metric." Software Engineering Journal 3.2 (1988): 30-36.