Software Vulnerability Fuzz Testing: A Mutation-Selection Optimization Systematic Review

Fatmah Yousef Assiri
Software Engineering Department, College of Computer Science and Engineering, University of Jeddah, Saudi Arabia
fyassiri@uj.edu.sa

Asia Othman Aljahdali
Cybersecurity Department, College of Computer Sciences and Engineering, University of Jeddah, Saudi Arabia
aoaljahdali@uj.edu.sa (corresponding author)

Received: 14 April 2024 | Revised: 6 May 2024 | Accepted: 14 May 2024

 DOI: https://doi.org/10.48084/etasr.6971

ABSTRACT

As software vulnerabilities can cause cybersecurity threats and have severe consequences, it is necessary to develop effective techniques to discover such vulnerabilities. Fuzzing is one of the most widely employed approaches that has been adapted for software testing. The mutation-based fuzzing approach is currently the most popular. The state-of-the-art American Fuzzy Lop (AFL) selects mutations randomly and lacks knowledge of mutation operations that are more helpful in a particular stage. This study performs a systematic review to identify and analyze existing approaches that optimize the selection of mutation operations. The main contributions of this work are to draw attention to the importance of mutation operator selection, identify optimization algorithms for mutation operator selection, and investigate their impact on fuzzing testing in terms of code coverage and finding new vulnerabilities. The investigation shows the effectiveness and advantages of optimizing the selection of mutation operations to achieve higher code coverage and find more vulnerabilities.

Keywords-software testing; software security; fuzz testing; vulnerabilities; mutation operator selection

I. INTRODUCTION

Nowadays, many software developers depend on open-source libraries when implementing software. Software vulnerabilities in such libraries can cause cybersecurity threats and have severe consequences [1]. The software testing phase requires considerable time, effort, and cost, particularly when there are many faults [2]. Therefore, there is a need to develop effective techniques to discover software security flows. Fuzzing is one of the most popular methods that has been adapted for software testing, such as kernel testing [3], firmware testing [4], and protocol fuzzing [5]. Fuzz testing is an automated software testing approach used to generate random data called "fuzz" that is used as a test case to discover vulnerabilities in the software under testing. Figure 1 illustrates the general fuzzing process, which consists of three main components: the test case generator, the executor, and the monitor. The generator is responsible for generating a new test case and providing the executor with new inputs. The executor runs the target programs utilizing these inputs. The monitor component tracks the execution to detect new defects or crashes. The objective of any fuzzer is to discover software defects [6]. The monitor collects specific run-time information, such as path coverage information, and passes it to the generator to guide its test-case generation process. When the target program crashes or reports errors, the bug detector module collects and analyzes related information to decide whether a bug exists. Eventually, vulnerabilities are filtered from all reported bugs through a bug filter [7].

There are general systematic reviews of fuzz testing, focusing on unifying the process and identifying gaps and solutions [7, 8 9, 10]. Other reviews shed light on studies that apply machine learning algorithms [11, 12], and some studies aimed to review fuzz testing on IoT firmware [13]. This work is a novel systematic review that focuses on optimization methods for mutation operator selection in mutation-based fuzzing, covering the period between 2017 and 2022. The main contributions of this paper are:

- Draw attention to the importance of mutation operator selection in fuzzing testing.
- Identify optimization algorithms for mutation operator selection in mutation-based fuzzing.

- Investigate the impact of optimization algorithms on fuzz testing in terms of code coverage and finding new vulnerabilities compared to the state-of-the-art AFL.
- Recommend potential future work.

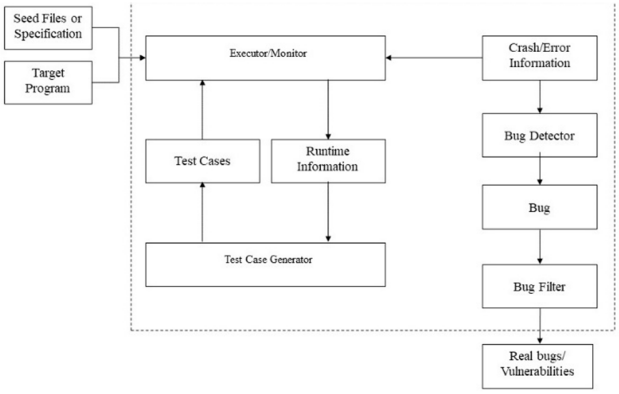

Fig. 1. Fuzzing process.

II. BACKGROUND

A. Fuzz Testing Types

Concerning input generation, fuzzing can be divided into generation-based and mutation-based. Generation-based fuzzing creates the test case from scratch and requires knowledge of the specifications that describe the file format or the network protocol. This category encompasses several works, including the use of constraint logic programming [14, 15]. Mutation-based fuzzing starts with a seed set (corpus) as a baseline and then mutates existing inputs to obtain new test cases. The seed is mutated utilizing mutation operations, including bit flip, byte flip, arithmetic increments, decrements of integers, etc. Examples of such fuzzers are AFL [16] and BuzzFuzz [17]. One of the advantages of mutation-based fuzzing is that it does not require knowledge of the software or protocol under testing. Thus, it is easier to start and widely employed. There are three main types of fuzzing depending on the use of the program source code: white-box, gray-box, and black-box. White-box fuzzers generate test cases by analyzing the internal program specifications and its execution. It utilizes the symbolic execution technique to enumerate interesting program paths using program analysis. Existing white-box fuzzing tools include KLEE [18] and SAGE [19]. Gray-box fuzzers produce seed input by obtaining some information about the internal specification code of the program under test and its executions through lightweight static analysis and dynamic information to gather information such as code coverage. Seeds are then mutated to generate new test case inputs. Generated test case inputs that cover new control locations are added to the seed corpus. Thus, code coverage is increased. Examples of such fuzzers include AFL, LibFuzzer, and Honggfuzz. Black-box fuzzing generates inputs without any knowledge of the program under test but observes its input/output. It is also called "data-driven" testing. Many fuzzers fall into this category, including [20, 21]. Table I shows examples of each fuzzing type.

TABLE I. EXAMPLES OF WHITE-BOX, GRAY-BOX, AND BLACK-BOX FUZZING

Fuzz tones	White-box	Gray-box	Black-box
Generation-based	Spike, Sulley, Peach		
Mutation-based	Miller	AFL, Driller, Vuzzer, TaintScope, Mayhem	SAGE, Libfuzzer

B. Fuzz Testing and Optimization

There are two methods to optimize fuzz testing: seed selection and mutation selection. Seeds are the inputs to the fuzzer that are deployed to create inputs to test the Software Under Test (SUT). Seeds are valid inputs in the space of inputs for SUT. An unlimited number of seeds can be used. However, the best seeds are those that generate test inputs and provide better code coverage, thus detecting system vulnerabilities. As selecting seeds can be a tedious process, optimization algorithms are applied to select seeds that lead to better test inputs. Particle Swarm Optimization (PSO) has been employed to improve seed selections in target-oriented fuzzing by computing the distance between the executed block and the target block where the suspected vulnerable point is located [22]. Control flow and function call graphs are statistically generated to compute the distance. Then, PSO and the format of all created inputs are changed to generate better inputs that are closer to the target point. This approach generates more inputs that reach the target point.

In [23], a multi-objective optimization model, MooFuzz, was developed to select the optimal seed set applying a non-dominated sorting algorithm. The main components of MooFuzz are the static analyzer, feedback collector, seed scheduler, and power scheduler. Static analyzers mark suspected locations and collect risk information, such as path risk and path frequency. Then, the feedback collector adjusts seed risk values to guide the seed scheduler, which uses the optimization model to prioritize seeds. Finally, the power scheduler assigns energy to seeds based on feedback information. A high-quality seed is more likely to be mutated and should be given more energy. Seed selection has been improved using heuristics to generate seeds that lead to valid test inputs to detect problems in newly visited regions [24]. The first heuristic collects both coverage information and execution information, and the second heuristic creates a string dictionary consisting of string constants that guide the fuzzer to create valid inputs. The performance of this approach is competitive with that of other fuzzers in terms of bug detection. On the other hand, mutation-selection optimization efficiently selects mutation operators by applying optimization algorithms. Mutation operations are implemented by the fuzzer to change the seeds to generate test cases that can detect anomalies in the system. This study investigates the effectiveness of mutation operation optimization methods to improve the fuzzing process.

III. RELATED WORKS

This study searched for secondary studies (reviews) that focused on mutation operation selection optimizations. Review papers concentrating on fuzz testing in general, such as [7, 9, 10, 12, 25], were excluded. Related papers should cover

methods and techniques to improve the selection of mutation operators in fuzz testing. Search engines were engaged, adding the terms "survey" or "review" to the search terms. There were no systematic review papers found on the selected topic.

IV. METHOD

This systematic literature review is based on well-established guidelines for conducting systematic reviews [26]. The process started by identifying and evaluating secondary studies that discuss the topic of mutation-selection optimizations for fuzz testing. Then, related surveys were analyzed to identify gaps and limitations. Based on the identified research gap, a set of research questions was developed to serve as the objectives of the review. The search engines and online libraries used to locate the relevant primary studies were identified and then a set of queries was defined to perform the search and the backward snowballing process was followed in the study identification process. Furthermore, inclusion and exclusion criteria as well as the quality assessment criteria used were defined for the primary studies.

V. RESEARCH QUESTIONS

The Research Questions (RQs) for this review were as follows:

- RQ1: What mutation operations are applied in studies of gray-box fuzzing optimization?
- RQ2: What techniques are utilized to optimize mutation operation selection?
- RQ3: How optimization methods affect the fuzzing process in terms of code coverage and newly discovered vulnerabilities compared to the state-of-the-art AFL?

VI. RESEARCH METHODOLOGY

The search involved research articles published between 2017 and 2022 in well-known libraries and top conferences. Then, a set of criteria was developed to include and exclude research articles. Data were selected based on title and abstract screening and then based on full-text screening. Finally, data were extracted from all relevant articles.

A. Source Material and Search Strategy

Well-known libraries and some of the top 10 conferences for computer security and software engineering were examined to identify eligible studies. The data sources were: IEEE Xplore, ACM Digital Library, Springer Link, Security Symposium (USENIX), International Conference on Software Engineering (ICSE), European Symposium on Research in Computer Security (ESORICS), ACM ASIA Conference on Computer and Communications Security (ASIACCS), IEEE Symposium on Security and Privacy (IEEE S&P), International Symposium on Software Testing and Analysis (ISSTA), and the International Conference on Software Testing, Verification, and Validation (ICST). Furthermore, references to candidate studies were reviewed for any other articles that may be missed. To perform the search, the respective search engines and online libraries were queried by combining key search terms using the Boolean expression "and" and incorporating other synonyms utilizing "or". The following search terms were put into service for the identification of primary studies:

- Fuzzing AND optimization, gray box fuzzing AND optimization, mutation fuzzing AND optimization
- (Mutation OR gray box) AND optimization
- (Mutation fuzzing OR Gray-box fuzzing) AND (optimization OR optimizing)
- Mutation fuzzing AND (assessment OR assessing) OR (evaluation OR evaluating)
- Software fuzzing AND optimization, software fuzzing AND mutation
- Software fuzzing AND mutation AND (assessment OR assessing) OR (evaluation OR evaluating).

As search capability limitations in some engines did not allow to follow the criteria, the terms "fuzzing," "fuzz," and "Fuzzer" were used.

B. Inclusion and Exclusion Criteria

The application of a set of inclusion and exclusion criteria is critical to filter related studies and keep the research focused on its main objectives. As this study focuses on optimizing mutation selection in fuzz testing, the following criteria were defined for inclusion:

- Studies that focus on the evaluation of mutation-based fuzzy selection
- Studies that use mutation-based fuzzing
- Studies applying optimization methods for better mutation selection
- Peer-reviewed articles published between 2017 and 2022.

The exclusion criteria were:

- Studies on fuzzing approaches that did not use mutation-based fuzzing (e.g., generation-based fuzzing)
- Studies that focused on seed selection optimization
- Studies with a primary focus on the points mentioned in the inclusion criteria but not yet published in a scientific conference proceeding or a journal.

C. Study Selection

After collection, the list of papers was finalized by removing duplicates. Each paper's title, abstract, and introduction were then screened to determine their suitability as primary or secondary papers following the criteria. Due to the use of different terminology, it was difficult to decide whether to entail each paper. Thus, the introduction and the proposed approach section were screened. Finally, 105 papers were involved, 88 of which were primary papers. Figure 2 portrays their distribution per year and source.

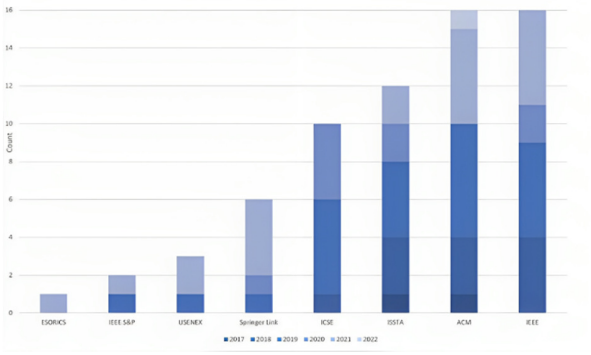

Fig. 2. Distribution of search results per data source.

D. Data Extraction and Quality Assessment

Data were extracted from the selected papers through full-text screening based on the research questions. For each paper, information on the mutation operators, optimization algorithms, dataset used, and evaluation results was collected in terms of coverage and finding new vulnerabilities.

VII. RESULTS AND DISCUSSION

Of the 88 primary papers, 56 were excluded. Eight of them could not be accessed through the selected institution, and the rest were outside the scope of this study. The selected articles were summarized based on the optimization method depicted in Figure 3.

The first group encompassed machine learning, deep learning, and reinforcement learning algorithms to optimize the selection of mutation operations. The study in [27] extends AFL by learning the probability distribution of its mutation operators on a program-by-program basis. In general, AFL randomly selects the operators to generate new inputs. The distribution of the sampling mutation operators was calculated using training programs, which significantly improved AFL performance. Additionally, Thompson sampling was applied, which is a bandit-based optimization approach that fine-tunes the mutator distribution. In [28], a neural network was applied to guide the selection of a position in the file whose value should be replaced by another random value. This approach found bugs in the PDF parser that were not detected before and covered more instructions than other random approaches. However, it also missed more instructions. In [29], a neural network was also deployed to develop an optimizing fuzzing process called NEUZZ, which learns the behavior of the program branching. Then, a gradient-guided optimization technique was developed to determine the bytes with the highest gradient values to mutate them, creating mutated inputs that detect more bugs of different types compared to other well-known fuzzers. DeepHunter is another fuzz-testing framework that employs deep neural networks [30]. The idea is to use a metamorphic mutation strategy to generate valid inputs with the same semantics as the original seed. It depends on domain knowledge to develop a mutation strategy that allows for generating semantic-preserved tests with few false positives. Fuzz testing was also addressed as a reinforcement problem adopting the Markov decision process to select mutation operators that improve a defined reward value to maximize code coverage and/or processing time [31-33]. MCMSFuzzer [33] optimizes the selection of mutation location, mutation intensity, and mutation algorithm.

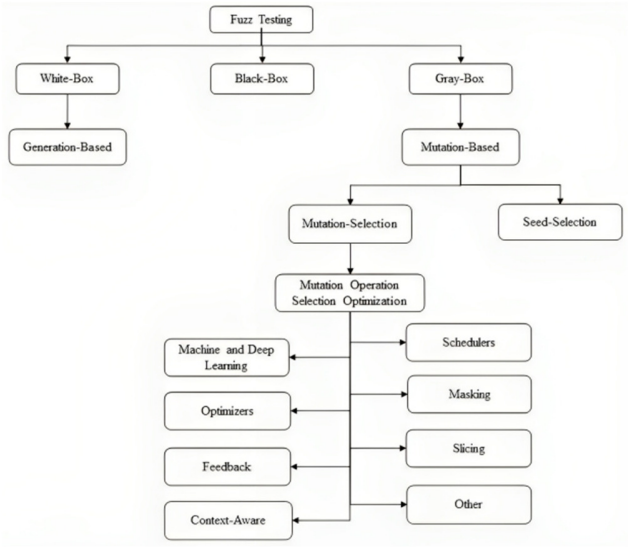

Fig. 3. The taxonomy of fuzz testing.

The second group applied optimization algorithms, such as particle swarm and ant colony optimizations, to further optimize the selection. MOpt is an optimized mutation selection strategy that applies PSO to find mutation operators that will maximize the effectiveness of fuzzers in terms of discovering vulnerabilities and execution time [34]. It was implemented on top of a well-known fuzzer, discovering more crashes, some of them being unique. PSO was also put into service for fuzzing Wi-Fi devices [35], providing a higher probability of mutation operators to maximize a cost function. Some fuzzing techniques focus on a recently changed code, as it was observed that such a code has more problems than the old code [36]. Ant colony optimization was applied at the byte level to assign higher probabilities to the code with higher impact to generate better-mutated inputs.

Feedback approaches, such as evolutionary algorithms, have been applied to optimize mutation operation selection. VUzzer is an evolutionary fuzzer that creates a feedback loop based on the control and data flow features of previous executions to determine where and how to mutate the input to create new/better inputs [37]. It deploys crossover to generate new inputs from the initial seeds and then mutate the input bytes. Knowledge-based evolutionary algorithms were developed to generate mutated inputs to detect more vulnerabilities. LearnAFL [38] is based on the equivalence class-based format generation theory, which learns the format of valid inputs and guides the random mutation process to create valid inputs that can execute new deep paths and successfully detect more unique faults. PerfFuzz [39] employs multidimensional feedback to produce inputs that exercise distinct locations in a program with a longer execution path, and it was implemented on top of AFL. To speed up the process and discover more paths, an interruptible mutation

method was followed in [40], which monitors the mutation input execution status and mutates those that execute more paths. A locality method was also proposed to determine the mutations that cause a significant change.

The SIVO fuzzer [41] uses parameterization and optimization engines, which dynamically select among different strategies to optimize their parameters for the target program based on the observed coverage. In [42], power scheduling was proposed to prioritize mutations, performing AST at the node level to identify nodes that help with bug detection. Mutation operators change the AST by replacing subtrees and combining bit-flip arithmetic mutations. An energy value is assigned to each node reflecting its rarity and influence, guiding the fuzzer to the most important mutations. In [43], a validity-based power scheduling technique was developed to assign higher energy to valid inputs, which doubled the energy for inputs with a validity of 50% and more. Granularity-aware mutator scheduling was proposed in [44]. This approach works by dynamically assigning ratios to different mutation operators, implemented in the AFL. It overcomes the limitations of AFL, which randomly selects mutators and deterministically specifies the number of each mutator, lacking the knowledge of mutators that are more helpful at a particular stage.

FairFuzz [45] implements a mutation mask strategy that optimizes mutation operation selection to reach locations that were not previously executed. It starts by selecting inputs that execute rare locations. Then, to generate mutated input, the mask strategy applies three mutation operators to a specified position. The byte position is selected from a pool of positions that will execute target locations. This strategy increased the percentage of mutated inputs that cover the target location. In [46], a mutation fuzzer was proposed that utilized forward dynamic slicing to identify program instructions influenced by a particular input byte. This information is employed to limit the mutation to only a subset of input bytes. The evaluation indicated that focused AFL improved bug discovery compared to vanilla AFL. In [47], a selection mutation algorithm was suggested based on a partition weight table to improve the fuzzing process to detect security issues in the NGAP protocol of the core 5G network. This approach calculates the probability of triggering anomalies in the target network before and after applying the proposed algorithms. The calculation demonstrates an increase in the probability of discovering anomalies. In [48], a new fuzzer was proposed to increase branch coverage based on context that explores different internal states. CMFuzz [49] is another context-aware adaptive mutation scheme that employs a contextual bandit algorithm (LinUCB) to select optimal mutation operators for various seed files. CMFuzz accomplishes this by dynamically extracting and encoding file characteristics, allowing mutation-based fuzzers to perform context-aware mutations. This study implemented CMFuzz on top of several fuzzers, including PTfuzz, AFL, and AFLFast, called CMFuzz-PT, CMFuzz-AFL, and CMFuzz-AFLFast, respectively.

Other approaches recommended the optimization of mutation selection. The TIFF prioritized mutation [50] infers the type of input by tagging each offset with a basic type. Types are inferred by identifying the data structure and performing a dynamic taint analysis, which produces high-quality inputs to discover memory-corruption bugs. In [51], a gray-box fuzzing approach (AFLCAI), which applied an optimization strategy for nondeterministic mutation, was introduced . AFLCAI uses an effector map mechanism to approximate metadata and improve code coverage through heuristic-guided mutation. JQF [52] is a Java platform to perform gray-box fuzz testing. In this approach, practitioners write QuickCheck-style test methods with formal parameters. JQF tests the bytecode of the target program using inputs generated in a coverage-guided fuzzing loop. Fuzz testing has been deployed to improve the security of autonomous vehicle systems. In [53], VulFuzz was presented, which is a fuzz testing framework that applies a weighting method to prioritize fuzz testing to the most vulnerable components engaging vulnerability metrics. As demonstrated in some studies, there are several advantages to the randomness of mutation selections. However, random strategies can damage the initial seeds. In [54], a hybrid mutation strategy was proposed that combined the random mutation strategy with a restricted mutation strategy. The restricted strategy applies to seeds that cover rare locations and the random strategy applies to other seeds. As a result, more branches are covered, leading to more crashes.

A. RQ1: What Mutation Operations are Applied in Studies of Gray-Box Fuzzing Optimization?

Mutation operators are one of the main components of fuzz testing. However, little attention has been paid to the group of selected operators. The literature review showed that mutation operators were not the focus of most studies, as operators were not mentioned at all in some studies. On the other hand, other studies built their approaches based on the state-of-the-art tool AFL, which consists of 11 operators including, but not limited to, byte flipping, inserting, deleting, replacing, and arithmetic increment and decrement. Several approaches are used to mutate seeds, including bit flipping, arithmetic mutation, and block-based mutation. Some fuzzers flip a fixed number of bits, while others randomly decide the number of bits to flip. Additionally, some fuzzers calculate the mutation ratio to determine the number of bit positions to flip. AFL contains another mutation by performing arithmetic mutation on a selected byte sequence from the seed, and the result is used to replace the selected byte sequence. As an illustration, AFL chooses a 4-byte value from a seed and considers it an integer i, then replaces the value in the seed with $i \pm r$, where r is a random integer such that $0 \leq r < 35$. Block-based mutation methodology works by inserting, deleting, replacing, or randomly permuting a randomly generated block into a random position of a seed [10].

However, some fuzz tests, such as AFL, are context-insensitive. In [48], some other approaches applied fuzzing to the AST. Thus, mutation operators that mutate the AST by inserting and replacing subtrees were applied, and other fuzzers added new operators that rearranged bytes [55]. In summary, most of the existing research focused on inserting, deleting, and replacing bytes or sequences of bytes by selecting random

values within a specified range. Table II summarizes the primary studies in terms of the optimization approach.

TABLE II. SUMMARY OF PRIMARY STUDIES.

Approach	References
ML and DL	ML [27], ANNs [28–30], Reinforcement [31–33]
Optimizer	PSO [34, 35], ACO [36]
Feedback	Evolutionary algorithm [37, 38], Multidimensional feedback [39], Parameter optimization [40, 41]
Scheduler	Power scheduling [42], Validity-based power scheduling [43], Granularity-aware scheduling [44]
Masking	Mask strategy [45]
Slicing	Forward dynamic slicing [46], Partition weight [47]
Context-aware	Context-aware [48], Contextual bandit algorithm [49]
Other	Heuristic mutation [51], Quick check test [52], Vulnerability metrics [53], Input type inference using dynamic taint analysis [50], Hybrid mutation [54]

B. RQ2: What Techniques are Utilized to Optimize Mutation Operation Selection?

A few studies have employed neural networks and Markov decision processes to guide the selection of mutation operations with better coverage and to detect more bugs. A neural network has been used to learn the probability distribution and behavior of the program branches to guide the selection process. Additionally, Thompson sampling has been deployed to fine-tune the mutator distribution. Well-known optimization algorithms, such as particle swarm and ant colony optimization have been utilized. These algorithms have been implemented to compute different probabilities and give a higher probability to the mutation operators that generate better-mutated input to detect anomalies and find more bugs.

Several mutation strategies have been followed to enhance the effectiveness of fuzzer tools in terms of finding more vulnerabilities by executing rare branches. Domain knowledge has been used to develop a mutation strategy that generates semantic-preserved input tests. Additionally, granularity-aware scheduling has been deployed to assign ratios to different mutation operations. A hybrid mutation strategy combined random mutation with a restricted mutation strategy. A multidimensional feedback mutation maximized execution counters for all program locations and then generated inputs that exercised distinct locations in a program. A mutation mask creation algorithm was applied to produce input, targeting rare branches in the code. An effector map mechanism has been employed to approximate metadata and improve both branch and path coverage. A selection mutation algorithm based on a partition weight table was applied to further improve the fuzzying process. Forward dynamic slicing and the interruptible mutation method were put into service to limit the mutation to a subset of the input bytes. The grammar-aware-trim strategy using AST employed this technique to create valid inputs. Additionally, the equivalence classes-based-format generation theory has been adopted to learn the format of valid inputs and guide the random mutation process to generate valid inputs that execute new paths to detect unique faults.

C. RQ3: How Optimization Methods Affect the Fuzzing Process in Terms of Code Coverage and Newly Discovered Vulnerabilities Compared to the State-of-the-Art AFL?

This section evaluates the studies mentioned based on their code coverage and newly discovered vulnerabilities. Code coverage is a measure that describes the degree to which the source code of a program has been tested. The basic coverage criteria are function, statement, and edge coverage, where edge coverage subsumes statement coverage. Software vulnerabilities cause system crashes, and adversaries could exploit them to cause serious security problems.

SIVO improved branch coverage by 20% over the best fuzzer and 180% over the AFL. It found more bugs compared to other fuzzers in 18 programs and found unique vulnerabilities in 11 benchmark programs. FairFuzz increased branch coverage faster than baseline AFL, and increased coverage by 10.6% in programs with nested condition structures. TIFF detected bugs much faster than existing solutions while increasing code coverage. In real-world applications, TIFF discovered bugs in half the time of state-of-the-art fuzzers. PerfFuzz generated inputs targeting the most-hit program branches from 5 to 69 times more than the existing solutions. Without affecting the running speed, AFLCAI increased branch coverage by 3.79%, while the new path was increased by 9.90%. Tuning the mutational operator distribution generated sets of inputs that yielded significantly higher code coverage and found more crashes faster and more reliably than both the baseline AFL and other AFL-based learning approaches. DeepHunter increased code coverage and discovered more bugs compared to AFL. JQF discovered 42 unknown bugs in OpenJDK, Apache Commons, and the Google Closure Compiler. MCMSFuzzer achieved higher code coverage and demonstrated a high capacity level in detecting vulnerabilities. VUzzer generated fewer inputs and succeeded in finding more crashes compared to AFL.

Tuning the mutational operator distribution generates inputs that lead to higher code coverage and more crashes faster than AFL and other AFL-based approaches. Mutation selection based on the partition weight table increases the probability of triggering anomalies compared to before using the algorithm. FairFuzz ran on nine different real-world benchmarks and achieved coverage above the confidence interval of other techniques. NEUZZ generated mutated inputs faster and accomplished higher edge coverage compared to the well-known fuzzers. When applying granularity-aware mutator scheduling, 12 new bugs and three new vulnerabilities (CVEs) were discovered. CMFuzz-based fuzzers showed better code coverage and found more crashes. Using Thompson sampling and Epsilon-Greedy algorithms in PTfuzz, CMFuzz-PT outperformed Thompson-PT in terms of unique crashes and paths. Compared to Greedy-PT, CMFuzz-PT increased the number of unique crashes and paths by 1.11 and 1.05, respectively. Superion improved code coverage over AFL by 16.7% in line coverage and 8.8% in function coverage and detected 31 new bugs. DeepFuzzer took more time to run but increased path coverage by 24%, branch coverage by 18%, and the number of unique triggered crashes by 123%. AFLTurbo discovered more paths than the baseline fuzzer and found 124

unique bugs, while AFL, AFL-Fast, and FairFuzz only found 8, 4, and 20 bugs, respectively.

Analyzing the previous studies in terms of code coverage and discovered vulnerabilities exhibited that SIVO, FairFuzz, TIFF, AFLCAI, MCMSFuzzer, DeepHunter, Superion, NEUZZ, and AFLTurbo improved code coverage and detected more bugs in the tested programs. In terms of execution time, TIFF found bugs faster than existing solutions, NEUZZ generated mutated inputs faster, and MOPT had a shorter execution time. However, DeepFuzzer took more time to run. Thus, it can be stated that all the above-mentioned studies succeeded in increasing code coverage and discovering more bugs in the tested programs compared to the state-of-the-art AFL and other existing tools. This confirms the effectiveness and advantages of optimizing the selected mutation operation to ameliiorate fuzz testing toward higher code coverage and spotting more bugs.

VIII. FUTURE WORK

- Many of the existing approaches compared their performance with that of the state-of-the-art AFL tool. However, it cannot be concluded which technique is the best since no control study has applied these approaches to the same set of data under the same experimental design. The field will be benefited from a comprehensive evaluation study that will highlight the best approach.
- A few sets of algorithms, such as evolutionary, neural networks, particle swarm, and ant colony, were applied to optimize the selection of mutation operations. However, different algorithms can be investigated and their performance can be compared.
- Implementing an open-source library that combines several optimization algorithms used with AFL will be helpful and will serve the needs of the industry. Additionally, it will help researchers carry out more studies in the area and develop more mature and optimized tools.
- As observed, mutation operators play an important role in the fuzzing process. However, not much attention has been paid to the set of selected operators. More studies are needed to develop more mutation operators and explore their advantages for the entire fuzzing process.

IX. THREATS TO VALIDITY

One of the main threats to the validity of this study is the lack of coverage of all related articles. To mitigate this threat, well-established guidelines were followed to perform the systematic review [24]. This study was also expanded to cover well-known libraries and some of the top 10 conferences for computer security and software engineering. Another threat is related to the terms utilized in the search process. All possible terms for mutation-based fuzz testing, such as gray-box and mutation fuzzing were identified. In some datasets, the criteria could not be applied due to search engine limitations, so broad terms, such as "fuzzing," "fuzz," and "fuzzer" were employed to ensure the efficacy of the search process. To mitigate threats related to the screening process, inclusion and exclusion criteria were clearly identified and followed.

X. CONCLUSIONS

Fuzzing is one of the most widely used techniques to discover software vulnerabilities. The mutation-based fuzzing approach is one type of fuzzer. AFL selects mutation operations randomly and ignores the knowledge of mutators that would lead to interesting inputs. Although mutation operators are the core of generating inputs, little attention has been paid to the selection set of operators. As far as is known, there have been no previous studies that systematically evaluateed mutation operator selection optimization methods. Some algorithms have been proposed to improve the fuzzing process by optimizing the selection of mutation operations. This investigation demonstrates that all of the previously mentioned studies are comparable to each other, as all improved the code coverage of the tested program and discovered more bugs. However, DeepFuzzer takes more execution time than the others. In general, most of the proposed approaches either reduced execution times or maintained a steady speed. This area can be further improved by focusing on the set of selected operators and proposing new operators that could significantly affect the process. In addition, future studies could investigate different optimization algorithms to optimize the selection of operations.

REFERENCES

[1] M. N. A. Khan, A. M. Mirza, R. A. Wagan, M. Shahid, and I. Saleem, "A Literature Review on Software Testing Techniques for Smartphone Applications," *Engineering, Technology & Applied Science Research*, vol. 10, no. 6, pp. 6578–6583, Dec. 2020, https://doi.org/10.48084/etasr.3844.

[2] W. Alkaberi and F. Assiri, "Predicting the Number of Software Faults using Deep Learning," *Engineering, Technology & Applied Science Research*, vol. 14, no. 2, pp. 13222–13231, Apr. 2024, https://doi.org/10.48084/etasr.6798.

[3] D. Song *et al.*, "PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary," in *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2019, https://doi.org/10.14722/ndss.2019.23176.

[4] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," presented at the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, USA, 2019, pp. 1099–1114.

[5] S. Gorbunov and A. Rosenbloom, "AutoFuzz: Automated Network Protocol Fuzzing Framework," *International Journal of Computer Science and Network Security*, vol. 10, no. 8, pp. 239–245, 2010.

[6] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A Survey for Roadmap," *ACM Computing Surveys*, vol. 54, no. 11s, Jun. 2022, Art. no. 230, https://doi.org/10.1145/3512345.

[7] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the Art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018, https://doi.org/10.1109/TR.2018.2834476.

[8] M. Zalewski, "American Fuzzy Lop: A Security Oriented Fuzzer." Google, May 18, 2024, [Online]. Available: https://github.com/google/AFL.

[9] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, vol. 75, pp. 118–137, Jun. 2018, https://doi.org/10.1016/j.cose.2018.02.002.

[10] V. J. M. Manès *et al.*, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021, https://doi.org/10.1109/TSE.2019.2946563.

[11] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, Jun. 2018, Art. no. 6, https://doi.org/10.1186/s42400-018-0002-y.

[12] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu, "A systematic review of fuzzing based on machine learning techniques," *PLOS ONE*, vol. 15, no. 8, 2020, Art. no. e0237749, https://doi.org/10.1371/journal.pone.0237749.

[13] C. Zhang, Y. Wang, and L. Wang, "Firmware Fuzzing: The State of the Art," in *Proceedings of the 12th Asia-Pacific Symposium on Internetware*, Singapore, Apr. 2021, pp. 110–115, https://doi.org/10.1145/3457913.3457934.

[14] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, Vasteras, Sweden, Jun. 2014, pp. 725–730, https://doi.org/10.1145/2642937.2642963.

[15] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the Rust Typechecker Using CLP (T)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, NE, USA, Aug. 2015, pp. 482–493, https://doi.org/10.1109/ASE.2015.65.

[16] M. Zalewski, "AFL: American Fuzzy Lop." [Online]. Available: https://lcamtuf.coredump.cx/afl/.

[17] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *2009 IEEE 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009, pp. 474–484, https://doi.org/10.1109/ICSE.2009.5070546.

[18] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *8th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, USA, Dec. 2008, pp. 209–224.

[19] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing," *Network and Distributed System Security (NDSS)*, vol. 8, pp. 151–166, 2008.

[20] D. Aitel, "An Introduction to SPIKE, the Fuzzer Creation Kit."

[21] S. Hocevar, "Zzuf: Application Fuzzer." [Online]. Available: https://github.com/samhocevar/zzuf.

[22] C. Chen, H. Xu, and B. Cui, "PSOFuzzer: A Target-Oriented Software Vulnerability Detection Technology Based on Particle Swarm Optimization," *Applied Sciences*, vol. 11, no. 3, Jan. 2021, Art. no. 1095, https://doi.org/10.3390/app11031095.

[23] X. Zhao, H. Qu, W. Lv, S. Li, and J. Xu, "MooFuzz: Many-Objective Optimization Seed Schedule for Fuzzer," *Mathematics*, vol. 9, no. 3, Jan. 2021, Art. no. 205, https://doi.org/10.3390/math9030205.

[24] Y. Fu, S. Tong, X. Guo, L. Cheng, Y. Zhang, and D. Feng, "Improving the Effectiveness of Grey-box Fuzzing By Extracting Program Information," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Guangzhou, China, Dec. 2020, pp. 434–441, https://doi.org/10.1109/TrustCom50675.2020.00066.

[25] R. Shakya and A. Rahman, "A preliminary taxonomy of techniques used in software fuzzing," in *Proceedings of the 7th Symposium on Hot Topics in the Science of Security*, Jun. 2020, https://doi.org/10.1145/3384217.3384219.

[26] B. Kitchenham, "Procedures for Performing Systematic Reviews," Keele University, Technical Report TR/SE-0401, Jul. 2004.

[27] S. Karamcheti, G. Mann, and D. Rosenberg, "Adaptive Grey-Box Fuzz-Testing with Thompson Sampling," in *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, Toronto, Canada, Jan. 2018, pp. 37–47, https://doi.org/10.1145/3270101.3270108.

[28] P. Godefroid, H. Peleg, and R. Singh, "Learn&Fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana, IL, USA, Oct. 2017, pp. 50–59, https://doi.org/10.1109/ASE.2017.8115618.

[29] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: Efficient Fuzzing with Neural Program Smoothing," in *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2019, pp. 803–817, https://doi.org/10.1109/SP.2019.00052.

[30] X. Xie *et al.*, "DeepHunter: a coverage-guided fuzz testing framework for deep neural networks," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Beijing, China, Apr. 2019, pp. 146–157, https://doi.org/10.1145/3293882.3330579.

[31] K. Böttinger, P. Godefroid, and R. Singh, "Deep Reinforcement Fuzzing," in *2018 IEEE Security and Privacy Workshops (SPW)*, San Francisco, CA, USA, 2018, pp. 116–122, https://doi.org/10.1109/SPW.2018.00026.

[32] Z. Zhang, B. Cui, and C. Chen, "Reinforcement Learning-Based Fuzzing Technology," in *Innovative Mobile and Internet Services in Ubiquitous Computing*, 2021, pp. 244–253, https://doi.org/10.1007/978-3-030-50399-4_24.

[33] H. Xu, B. Cui, and C. Chen, "Fuzzing with Multi-dimensional Control of Mutation Strategy," in *Innovative Mobile and Internet Services in Ubiquitous Computing*, Asan, Korea (South), 2022, pp. 276–284, https://doi.org/10.1007/978-3-030-79728-7_27.

[34] C. Lyu *et al.*, "MOPT: Optimized Mutation Scheduling for Fuzzers," presented at the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, USA, 2019, pp. 1949–1966.

[35] M. E. Garbelini, C. Wang, and S. Chattopadhyay, "Greyhound: Directed Greybox Wi-Fi Fuzzing," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 2, pp. 817–834, Aug. 2022, https://doi.org/10.1109/TDSC.2020.3014624.

[36] X. Zhu and M. Böhme, "Regression Greybox Fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, Aug. 2021, pp. 2169–2182, https://doi.org/10.1145/3460120.3484596.

[37] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *NDSS Symposium 2017*, San Diego, CA, USA, Feb. 2017, https://doi.org/10.14722/ndss.2017.23404.

[38] T. Yue, Y. Tang, B. Yu, P. Wang, and E. Wang, "LearnAFL: Greybox Fuzzing With Knowledge Enhancement," *IEEE Access*, vol. 7, pp. 117029–117043, 2019, https://doi.org/10.1109/ACCESS.2019.2936235.

[39] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Amsterdam, Netherlands, Apr. 2018, pp. 254–265, https://doi.org/10.1145/3213846.3213874.

[40] L. Sun, X. Li, H. Qu, and X. Zhang, "AFLTurbo: Speed up Path Discovery for Greybox Fuzzing," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2020, pp. 81–91, https://doi.org/10.1109/ISSRE5003.2020.00017.

[41] I. Nikolić, R. Mantu, S. Shen, and P. Saxena, "Refined Grey-Box Fuzzing with Sivo," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2021, pp. 106–129, https://doi.org/10.1007/978-3-030-80825-9_6.

[42] J. Deng, X. Zhu, X. Xiao, S. Wen, Q. Li, and S. Xia, "Fuzzing With Optimized Grammar-Aware Mutation Strategies," *IEEE Access*, vol. 9, pp. 95061–95071, 2021, https://doi.org/10.1109/ACCESS.2021.3093904.

[43] V. T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart Greybox Fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, Sep. 2019, https://doi.org/10.1109/TSE.2019.2941681.

[44] L. Situ, L. Wang, X. Li, L. Guan, W. Zhang, and P. Liu, "Energy Distribution Matters in Greybox Fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, Montreal, Canada, May 2019, pp. 270–271, https://doi.org/10.1109/ICSE-Companion.2019.00109.

[45] C. Lemieux and K. Sen, "FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Montpellier, France, Jun. 2018, pp. 475–485, https://doi.org/10.1145/3238147.3238176.

[46] U. Kargén and N. Shahmehri, "Speeding Up Bug Finding using Focused Fuzzing," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, Hamburg, Germany, May 2018, pp. 1–10, https://doi.org/10.1145/3230833.3230867.

[47] Y. Hu, W. Yang, B. Cui, X. Zhou, Z. Mao, and Y. Wang, "Fuzzing Method Based on Selection Mutation of Partition Weight Table for 5G Core Network NGAP Protocol," in *Innovative Mobile and Internet Services in Ubiquitous Computing*, Asan, Korea (South), 2022, pp. 144–155, https://doi.org/10.1007/978-3-030-79728-7_15.

[48] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *2018 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, May 2018, pp. 711–725, https://doi.org/10.1109/SP.2018.00046.

[49] X. Wang, C. Hu, R. Ma, D. Tian, and J. He, "CMFuzz: context-aware adaptive mutation for fuzzers," *Empirical Software Engineering*, vol. 26, no. 1, Jan. 2021, Art. no. 10, https://doi.org/10.1007/s10664-020-09927-3.

[50] V. Jain, S. Rawat, C. Giuffrida, and H. Bos, "TIFF: Using Input Type Inference To Improve Fuzzing," in *Proceedings of the 34th Annual Computer Security Applications Conference*, San Juan, PR, USA, Sep. 2018, pp. 505–517, https://doi.org/10.1145/3274694.3274746.

[51] Z. Cai, H. Wang, and X. Qin, "A Heuristic Guided Optimized Strategy for Non-Deterministic Mutation," in *Proceedings of the 3rd International Conference on Computer Science and Application Engineering*, Sanya, China, Oct. 2019, https://doi.org/10.1145/3331453.3361295.

[52] R. Padhye, C. Lemieux, and K. Sen, "JQF: coverage-guided property-based testing in Java," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Beijing, China, Jul. 2019, pp. 398–401, https://doi.org/10.1145/3293882.3339002.

[53] L. J. Moukahal, M. Zulkernine, and M. Soukup, "Vulnerability-Oriented Fuzz Testing for Connected Autonomous Vehicle Systems," *IEEE Transactions on Reliability*, vol. 70, no. 4, pp. 1422–1437, Oct. 2021, https://doi.org/10.1109/TR.2021.3112538.

[54] J. Liang *et al.*, "DeepFuzzer: Accelerated Deep Greybox Fuzzing," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 6, pp. 2675–2688, Aug. 2021, https://doi.org/10.1109/TDSC.2019.2961339.

[55] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-Aware Greybox Fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, Montreal, Canada, May 2019, pp. 724–735, https://doi.org/10.1109/ICSE.2019.00081.

AUTHORS PROFILE

Fatmah Assiri is an associate professor in Software Engineering with a history of working in leading positions. He specializes in software quality and has experience with data and machine learning. Served as a consultant for the Entrepreneurship and Innovation Center of the University of Jeddah. Participated as a mentor and judge at local and international events and was invited speaker at many other events.

Asia Aljahdali is an associate professor in the Cybersecurity Department of the University of Jeddah. She received her Ph.D. in computer science from Florida State University in 2017 and an M.Sc. degree in information security in 2013. She has worked at King Abdul-Aziz University as an assistant professor. In 2020, in addition to her academic work, she worked as a cybersecurity consultant for the Cybersecurity Administration at Jeddah University. Her current research interests include data hiding, software security, IoT security, and malware analysis.