An Automata-Driven Machine Learning Framework for Intelligent Web Application Testing

Pawan.S[1], Swarna H.R[2]
[1]*Department of Computer Science and Engineering, Srinivas University Institute of Engineering and Technology Mukka*
[2]*Professor, Srinivas University Institute of Engineering and Technology Mukka*
doi.org/10.64643/IJIRTV12I12-206821-459

***Abstract*—The evolution of modern web applications towards being more dynamic and state-driven makes many traditional testing methods less efficient due to complexity of such applications. Traditional approaches to web testing often prove inefficient when dealing with multi-staged workflow, state change management, and various input-dependent application actions. This paper presents an adaptive approach to testing based on automata modeling and machine learning to facilitate adaptive testing of modern web applications. Firstly, this approach requires constructing an automata model describing possible web application navigation and interactions. On the basis of this model, systematic paths covering all available application states are derived. Machine learning is applied in order to identify the most problematic paths and optimize their selection for testing purposes. Also, the input generation process involves fuzzing techniques which allow producing unexpected and varied input values that increase the chance to detect potential vulnerabilities. The combination of automata modeling, machine learning, and fuzzing allows performing adaptive web application testing and ensures structural consistency. The presented approach provides better test coverage, fault detection ability, and decreased testing efforts in comparison with traditional methods.**

***Index Terms*—Automata-Based Testing, Machine Learning, Web Application Testing, Hybrid Testing Framework, Fuzzing, Intelligent Testing.**

I. INTRODUCTION

The evolution of web applications has led to the development of sophisticated and interactive applications that make use of workflow and asynchronous processing. Such applications consist of multiple user interactions and API calls that add to the complexity. Testing approaches such as rule-based automation and manual testing prove to be inadequate in many cases and hence fail to detect logic errors, security vulnerabilities, and issues related to transitions between states. It is clear from recent literature on testing techniques for web applications that traditional techniques tend to miss navigation paths and state-sensitive aspects of the application.

From the groundwork carried out on automata-based testing, it becomes clear that finite-state machines and other formal approaches can be used effectively to create test data sets and test execution plans. Automata theory shows how grammars can be used to represent navigation paths and interactions, thus leading to a better way to develop tests that cover more state space. Nevertheless, there exists evidence from literature that suggests that approaches based on automata theory lack scalability and must be updated frequently to keep up with the evolving nature of web applications.

Alongside improvements in machine learning, there have been developments in adaptive testing approaches that predict fault-prone paths and prioritize test executions. Machine learning algorithms can be utilized in software testing to examine execution traces and anomalies and optimize regression tests. In this manner, there is an element of adaptability and intelligence in testing that allows the testing system to learn from its previous results. Nonetheless, research has demonstrated that machine learning testing does not offer structural assurance and may lead to the creation of invalid test cases without structural input.

Moreover, investigations into fuzzing algorithms have illustrated their utility in generating multiple types of test data and uncovering unforeseen faults. Grammar-based fuzzing and coverage-guided fuzzing strategies

have been shown to enhance their ability to explore further into execution paths and detect potential security issues. Nevertheless, one of the significant drawbacks of fuzzing algorithms is that they usually create irrelevant and semantically invalid test cases, which reduces their effectiveness.

Apart from the testing problems mentioned above, modern web applications are very susceptible to common security risks like SQL injection, cross-site scripting (XSS), authentication attacks, and parameter tampering. These risks take advantage of loopholes in application input validation and handling of sessions and parameters. According to existing research, it is important to ensure security awareness in testing through simulation of such attacks.

From the studies conducted there is a suggestion that automata testing is used to provide structure correctness; machine learning is used to enable adaptability, while fuzzing allows generation of various inputs. However, most of the existing methods are independent and do not incorporate all these features. This motivates the need to develop a hybrid approach for testing which uses all the three components.

An integrated framework based on automata and machine learning is presented as a solution for automated testing of web applications. The framework builds finite state automation models capturing workflows in web applications and creates test paths in a systematic manner. Attack scenarios like SQL injection, XSS, authentication attacks, and parameter tampering are considered in test path creation.

A Decision Tree machine learning model is then used to evaluate test outcomes, classify potential vulnerabilities, and prioritize high-risk execution paths. This integrated approach enhances test coverage, improves vulnerability detection, and reduces manual testing effort.

II. LITERATURE REVIEW

Web application testing has attracted extensive research interest utilizing formal modeling, machine learning, fuzzing, and combined methodologies. In this section, we discuss the existing literature on test methods based on automata, machine learning, fuzzing, and intelligent hybrid testing.

A. Automata-Based Web Application Testing

Automata-based testing is commonly employed for creating models to represent application behavior and developing structured tests. The work done by Andrews et al. [1] in finite state machine (FSM) modeling for web applications has proven the potential of the use of automata-based modeling in developing systematic tests. Additionally, structural testing techniques for web applications have been discussed by Ricca and Tonella [17] and Liu et al. [18], proving that through formal modeling, faults related to navigation are discovered easily.

Further development in automata-based testing was performed by Zozulya et al. [3] in proposing a framework for systematic test case generation using finite automata. Further progress was achieved in Chen et al.'s work [9] where test cases were generated based on sequence diagrams and automata models, which improved multi-step workflow representation. Another proof that model-based testing increases fault detection capabilities in web applications has been provided by Garousi et al. [4]. A number of recent research papers have been published about learning-based automata and hybrids. For example, Di et al. [7] presented their research on the use of learning automata for testing based on dynamic modification of testing behavior. Another paper [19], written by Shah and Mughal, examined artificial intelligence-based automata approaches and proved that intelligent automata-based models allow for more effective testing. Furthermore, Gujar et al. [11] mentioned the use of finite automata for structured pattern validation.

Thus, it is clear that testing using automata provides more structured testing and better test coverage. Nevertheless, the literature shows that automata-based testing still needs to be manually modeled without adaptive intelligence.

B. Fuzzing-Based Testing Approaches

Fuzzing is widely applied to generate different inputs and detect vulnerabilities within web applications. Specifically, Bozic and Wotawa [20] developed planning-based security testing via attacks grammars, showing better results regarding vulnerability detection. Moreover, van Rooij et al. [12] created webFuzz – a grey-box fuzzing framework to explore states of web applications.

Recently, the application of machine learning techniques to fuzzing received significant attention. The effectiveness of grammar-based and coverage-guided fuzzing approaches was systematically analyzed by Wang et al. [13] and Chen et al. [14]. Godefroid et al. [16] developed Learn&Fuzz – a machine learning-assisted approach to generate inputs for fuzzing.

Modern studies on the topic cover such approaches as automatic grammar detection and intelligent fuzzing. In particular, Song and Alves-Foss [21] developed an approach to automate the process of grammar detection, while Li et al. [22] proposed an intelligent input generation framework called JIMA-Fuzzing. Lin [24] proposed hybrid fuzzing utilizing language models and semantic feedback.

The effectiveness of fuzzing approaches can be seen from higher vulnerability detection and better testing diversity. However, the problem with fuzzing techniques is that they produce irrelevant or syntactically incorrect input data.

C. Machine Learning-Based Testing Approaches

Machine learning methods have been utilized extensively in software testing and vulnerability detection. Paduraru and Melemciuc [6] suggested the use of machine learning methods for automatic test data generation.

Appelt et al. [15] came up with machine learning-assisted evolutionary testing of web application firewalls that improved testing efficacy. Recent works have focused on deep learning and AI-assisted testing practices.

Naveed et al. [5] described large language models and their uses in software testing. Huang et al. [23] explored LLM-assisted fuzzing methods to discover vulnerabilities.

Kapula [10] and Kertusha et al. [8] reviewed the growing prevalence of AI-assisted web testing and demonstrated how machine learning is capable of enhancing adaptive testing and anomaly detection. These papers show how machine learning improves testing but does not provide any structural guarantees when performed separately.

D. Hybrid Testing Approaches

There are several hybrid testing approaches using automata, machine learning, and fuzzing. Jin et al. [26] introduced the concept of testing and automatic input generation for JavaScript-enabled web applications. In addition to that, Bertolino [25] highlighted future trends in software testing, pointing out the need for intelligent testing frameworks. Modern hybrid approaches include grammar-based fuzzing along with machine learning.

In particular, Eberlein et al. [2] suggested evolutionary grammars to improve input diversity. Moreover, learning-based automata and hybrid frameworks [7], [24] illustrate the advantages of a combination of modeling and adaptive algorithms.

According to these works, the use of hybrid testing frameworks enables better coverage, more efficient vulnerability detection, and adaptiveness. Nonetheless, currently there are no uniform frameworks using all three aspects together.

III. PROPOSED FRAMEWORK

Testing methodology for automatic testing of web applications that incorporates model-based automation using finite state machines to generate attack scenarios, and then uses machine learning for intelligent evaluation and testing.

The system is structured in such a way that it provides for systematic exploration of the behavior of the applications, simulation of attack scenarios, and detection of vulnerable execution paths.

The system operates through a series of connected modules, which include modeling with automata, generation of test paths, injecting attacks into the web application, and finally machine learning-based evaluation of the attacks.

A. Architecture Overview

The suggested framework adopts a structured pipeline approach where the web application is initially modeled using automata before generating test cases, simulating attacks, and applying evaluation through machine learning.

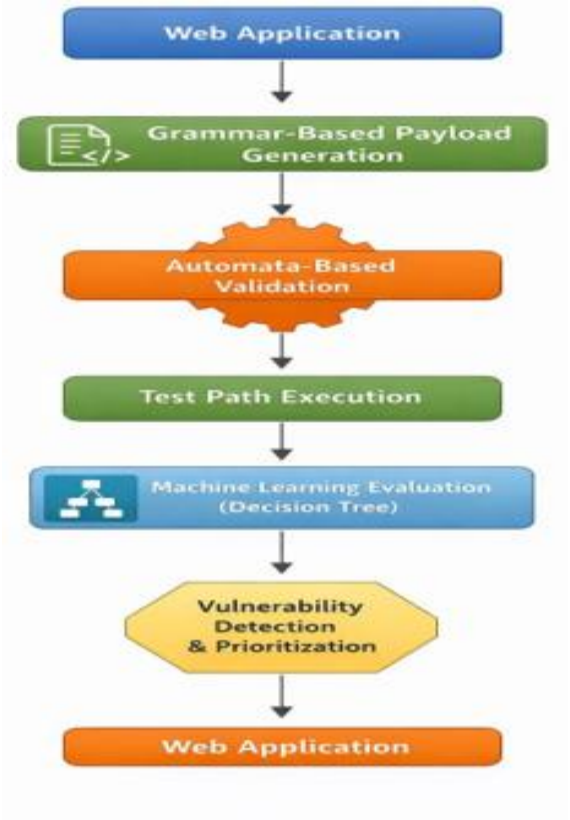

Fig. 1. Automata Based Threat Detection System Architecture

B. Grammar Based Attack Generation

Stage one of the framework involves creating structured payloads for an attack using predefined grammar rules. The grammar specifies patterns of the attack that can be done using a particular form of attacks like SQL injections and XSS.

For instance:

SQL Injection Grammar:

<query> ::= ' OR '1'='1

XSS Grammar:

<script> ::= <script>alert('XSS')</script> Unlike random fuzzing, grammar-based generation ensures that inputs follow realistic attack structures. This allows the system to simulate real-world attack scenarios more effectively.

C. Automata-Based Validation

Validation using automata is one of the most important processes in the framework. The produced inputs can be validated with finite-state automata (FSA) in order to confirm their compliance with certain characteristics and behaviors.

In such a framework: Each grammar rule is modeled by automata.Acceptance of a sequence is performed by the automaton.

Unnecessary inputs are rejected:For instance, the SQL injection automaton will check whether the input is consistent with the SQL injection attack pattern. The same procedure is applied for the XSS payloads as well.

D. Test Path Execution

The validated attack payloads are then inserted into the web application following specific testing routes. The test routes are generated based on various application flows including login, search, and transactions.

This phase facilitates Interaction with actual application elements,Monitoring of system reactions to attacks and Recording of system reactions.

E. Machine Learning-Based Evaluation

The last phase of the framework includes the use of the Decision Tree machine learning algorithm for test execution analysis and vulnerability classification.

The Decision Tree algorithm takes into account various features like:

1. Structure of input
2. Type of attack
3. Response behavior
4. Success or failure of execution

From these features, the algorithm infers:

a. If the input is malicious or not

b. The criticality level of vulnerability

The Decision Tree algorithm has been chosen because of its explainability and rule-based decision-making capabilities.

IV. METHODOLOGY

This section covers the actual realization of the proposed grammar-driven and automata-based testing methodology through explaining how each of its elements was realized using suitable technological means. The testing infrastructure will be developed using Python as the main programming language, which enables effective development of all required parts (automation, data processing, and machine learning) within a single language framework. The testing framework will feature a modular pipeline structure, combining grammar generation, automaton validation, execution, and learning-based evaluation

stages in an organic manner.

The process starts with the payload generation module based on grammar specifications. Such a module should produce structured data representing specific attack payloads. The realization of this module will involve usage of Python code defining templates according to grammar specifications for various attack types. Contrary to the random string's technique, this module creates attack payload patterns corresponding to correct syntactic constructions for particular attacks like SQL injections and cross-site scripting. Using such patterns, the module generates numerous samples based on specific algorithms.

After the creation of payloads, the automata-based validation is done via rule-based validation of payloads' states. Even though an automaton may not be directly created using third-party libraries, its functions are replicated by means of well-defined validation rules in Python. The validation process involves checking the existence and order of certain tokens in each payload, thereby replicating state transitions of a finite state machine. For example, the validation of SQL injection payloads involves the identification of logical operators and conditional statements in the payloads, while that of XSS payloads involves identifying tags. This module is essential for filtering out invalid inputs, thus minimizing the execution of irrelevant test cases.

Finally, the validated payloads are applied to the target web application using Selenium WebDriver, which acts as the main automation engine in this implementation. Through Selenium, the web application can be used in the same way as a normal user would use it. The Chrome web browser operates via Chrome Driver that facilitates the manipulation of elements, for example, login boxes, search boxes, and URL parameter values. When executing, payloads are delivered to the correct input fields, and the system checks the reaction of the application including any page change, errors, or outputs.

The data collected during the execution phase becomes large in volume, and this data is handled by Pandas and NumPy libraries to create a structured data set. In the dataset, each test case will be represented as an instance having some features taken from both input payload and application's responses. Features will include information such as type of input, type of attacks, and behaviors of the target system. For this purpose, Pandas helps in organizing and manipulating data efficiently while NumPy assists in processing numerical data for training purposes.

For creating the machine learning model, Scikit-learn library can be used. In particular, the implementation utilizes a Decision Tree classifier for vulnerability assessment. The training process involves preparing the dataset and feeding it into the model for it to learn the patterns present in both cases: vulnerabilities and non-vulnerabilities. Decision Tree classifier was chosen because of its efficiency and ease to interpret when predicting vulnerable payloads. After training, the created machine learning model will be able to classify newly provided test cases and flag high-risk inputs according to the decision rules learned during the training phase.

Each module is part of a coherent execution pipeline running inside the Python environment, enabling smooth flow of information between all modules. The process includes the following steps: payload generation, validation, automatic execution, data collection, and evaluation using machine learning. In such way, the modularity ensures that each module can operate independently but still plays an important role in the entire process of testing and makes the system capable of being extended and modified according to needs.

This solution proves that the proposed architecture is efficient in overcoming the weaknesses of existing methods and enables a structured and intelligent approach to testing web applications.

V. PROTOTYPE EVALUATION AND ANALYSIS

This proposed framework has been implemented in a prototype to analyze the feasibility of combining payload generation based on grammar, validation based on automata, and evaluation based on machine learning for testing web applications. This evaluation is carried out in a controlled setting by taking a sample web application that has typical user input methods like log-in forms, search boxes, and other parameterized inputs.

In order to understand the working and effectiveness of each module separately as well as their interaction in the pipeline, this analysis has been carried out. The findings of the prototype implementation have been presented in Table 1. This table presents a module-wise analysis of the framework along with their

functionality and observations made in the process of their execution.it can be seen that the grammar-based generator produced inputs, which represented realistic attack patterns; thus, the test was performed using valid inputs. The validation module, based on automata, effectively filtered out structurally invalid inputs in order to save time for unnecessary execution and improve the testing procedure's performance. The execution module was capable of interacting with the web application through automatic input injection. Meanwhile, the machine learning technique proved to have capability of analyzing execution results and classifying test cases according to their behavior.

Apart from evaluation of each individual module, the proposed framework was tested by measuring its capability of simulating various attacks against web applications. Below in Table 2, there is a list of attacks used in prototype testing, along with their respective inputs and behavior. Table 2 shows that the proposed framework is able to simulate several different types of attacks against the application. By using structured payloads generated through grammar rules, it was possible to interact with the application realistically and obtain response patterns by executing attack payloads. In other words, the framework proved its capabilities to effectively emulate attack cases and monitor system reactions.

TABLE 1: Module-wise Evaluation

Module	Function	Observation
Grammar Based Generator	Generates structured Payloads	Produced syntactically valid real world-like attacks
Automata Validator	Validates payload structure	Filter invalid inputs
Execution Engine	Injects payloads into web application	Successfully executed attack scenarios
ML Evaluation (Decision Tree)	Classifies test cases	Distinguish Suspecious and Normal behaviour

On the side of the analysis, the presented prototype proves the efficiency of the approach in contrast to traditional methods of fuzzing where unstructured test cases are randomly used. This method ensures that all the tested attack inputs are valid and syntactically correct because both of the grammar-based generator and automata validation check it beforehand. Moreover, the Decision Tree approach allows implementing a machine-learning based way to analyze response patterns and detect vulnerabilities.

The prototype presented above proves that the introduced framework may be efficiently used. However, currently, it is a small step towards the future research because there are many aspects that need to be elaborated in terms of the larger scale evaluation, different attack types, and machine learning algorithms

TABLE 2: Attack Generation and Observed Behaviour

Attack Type	Target Module	Observation
SQL Injection	Login Form	Application response captured, indicating input processing behavior
XSS	Search Field	Script handling behavior observed in response
Authentication Attack	Login System	Repeated Failure responses recorded
Parameter Tampering	URL Input	Altered response behaviour observed

VI. CONCLUSION

With the evolution of today's applications into increasingly sophisticated and complex systems, more advanced methods for testing such programs have become necessary. Traditional approaches, which include generating random test cases and manually checking their outcomes, are insufficient for finding potential flaws that emerge from dynamic processes within the program itself or from interactions performed by its users.

The described concept highlights how structured and intelligent approaches in the generation of test data and its subsequent validation contribute to the overall efficiency of the process. The use of grammar-driven algorithms guarantees that the payloads created by the program follow realistic patterns of malicious attacks,

and the validation process based on automata ensures the elimination of useless payloads and execution cases.

Finally, the inclusion of an automated execution process allows observing how the application behaves while executing attack scenarios. Moreover, adding a decision tree model to analyze execution results adds another layer of flexibility and intelligence to the testing process.

Prototype implementation of the approach ensures its applicability in designing such a pipeline. The system managed to perform multiple simulations of application-level attacks and capture their responses, which proves the practicality of the approach. From observations made during the experiments, it appears that input generation through structures along with validation through automata is more efficient than regular techniques.

This research paves the way to further study hybrid test frameworks. Potential future enhancements of the approach include adding additional types of grammars to support a variety of attacks, implementing more efficient automata validation techniques, and performing a massive-scale evaluation of real-world websites.

ACKNOWLEDGMENT

The authors express their sincere gratitude to the Department of Computer Science and Engineering, Srinivas University Institute of Engineering and Technology Mukka, for providing the necessary support and academic environment to carry out this work.The authors also thank all faculty members and peers for their encouragement and valuable suggestions during the course of this study.

Finally, the authors acknowledge all individuals who directly or indirectly contributed to the successful completion of this work.

REFERENCES

[1] A. A. Andrews, J. Offutt, and R. T. Alexander, “Testing web applications by modeling with FSMs,” Washington State University, George Mason University & Colorado State University, 2005.

[2] M. Eberlein, Y. Noller, T. Vogel, and L. Grunske, “Evolutionary grammar-based fuzzing,” Humboldt-Universität zu Berlin, 2020.

[3] M. M. Zozulya, O. J. Kravets, I. V. Atlasov, I. A. Aksenov, L. M. Bozhko, and P. A. Rahman, “Algorithmization of the software testing system based on finite automata,” International Journal of Information Technologies and Security, vol. 14, no. 1, 2022.

[4] V. Garousi, A. B. Keleş, Y. Balaman, and A. Arcuri, “Model-based testing in practice: An experience report from the web applications domain,” Kristiania University College, 2019.

[5] H. Naveed et al., “A comprehensive overview of large language models,” 2024.

[6] C. Paduraru and M.-C. Melemciuc, “An automatic test data generation tool using machine learning,” 2018.

[7] C. Di, S. Li, F. Li, and K. Qi, “A novel framework for learning automata: A statistical hypothesis testing approach,” 2019.

[8] I. Kertusha, G. Assres, O. Duman, and A. Arcuri, “A survey on web testing: On the rise of AI and applications in industry,” Kristiania University College, 2025.

[9] Z. Chen, Z. Duan, B. Yu, C. Tian, and M. Ding, “A test case generation approach based on sequence diagram and automata models,” 2016.

[10] K. Kapula, “A survey on web testing: On the rise of AI and applications in industry,” ICS Global Soft Inc., 2024.

[11] H. Gujar, K. Gawli, V. Ingle, S. Deokate, and Y. Chungade, “A review paper on finite automata application in string identification,” 2022.

[12] O. van Rooij, M. A. Charalambous, and D. Kaizer, “webFuzz: Grey-box fuzzing for web applications,” 2021.

[13] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu, “A systematic review of fuzzing based on machine learning techniques,” 2020.

[14] 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, 2018.

[15] D. Appelt, C. D. Nguyen, A. Panichella, and L.

[16] C. Briand, “A machine learning-driven evolutionary approach for testing web application firewalls,” 2016.

[17] P. Godefroid, H. Peleg, and R. Singh, “Learn&Fuzz: Machine learning for input fuzzing,” Microsoft Research, 2019.

[18] F. Ricca and P. Tonella, “Analysis and testing of web applications,” 2001.
[19] C.-H. Liu, D. C. Kung, P. Hsia, and C.-H. Tsu, “Structural testing of web applications,” 2001.
[20] A. J. Shah and M. H. Mughal, “Review on AI & DFA based practical approaches of automata theory,” 2024.
[21] J. Bozic and F. Wotawa, “Planning-based security testing of web applications with attack grammars,” 2020.
[22] J. Song and J. Alves-Foss, “A fuzzing tool based on automated grammar detection,” 2024.
[23] Y. Li et al., “A Fuzzing Tool Based on Automated Grammar Detection (JIMA-Fuzzing),” Software, vol. 3, no. 4, 2024.
[24] L. Huang, P. Zhao, H. Chen, and L. Ma, “Large language models based fuzzing techniques: A survey,” University of Sydney, 2024.
[25] S. Lin, “Hybrid fuzzing with LLM-guided input mutation and semantic feedback,” 2024.
[26] A. Bertolino, “Software testing research: Achievements, challenges, dreams,” Future of Software Engineering (FOSE), pp. 85–103, 2007.
[27] W. Jin, K. Sen, and A. Orso, “Automated testing and input generation for JavaScript web applications,” Proc. Int. Symp. Software Testing and Analysis (ISSTA), pp. 10–20, 2018.