Metamorphic Relation Prediction for Security Vulnerability Testing of Online Banking Applications

Karishma Rahman
Gianforte School of Computing
Montana State University, Bozeman MT, USA
karishma.rahman@student.montana.edu

Ann Marie Reinhold
Gianforte School of Computing
Montana State University, Bozeman MT, USA
Pacific Northwest National Laboratory, Richland WA, USA
reinhold@montana.edu

Clemente Izurieta
Gianforte School of Computing
Montana State University, Bozeman MT, USA
Pacific Northwest National Laboratory, Richland WA, USA
Idaho National Laboratory, Idaho Falls ID, USA
clemente.izurieta@montana.edu

***Abstract*—Software is essential in modern systems, and reliable testing of it is crucial due to the Oracle problem, which refers to the difficulties in distinguishing correct software behavior. Testing outputs from various inputs in online banking applications is complex and costly, making full automation necessary for efficiency and cost reduction. Metamorphic Testing (MT) addresses this by generating test inputs and evaluating outputs based on Metamorphic Relations (MRs), which dictate output changes with input modifications. However, identifying MRs has traditionally been manual and time-consuming. This paper presents an automated MT approach for online banking applications with vulnerabilities from the OWASP top 10. We created a prediction model using graph representations to automate MR detection, providing a catalog of 8 system-agnostic MRs for enhanced security testing. Results indicate that most MRs achieve prediction scores over 80%, demonstrating the practical effectiveness of this approach for improving online banking security through automated metamorphic testing.**

***Index Terms*—Metamorphic Testing, Metamorphic Relation, Vulnerabilities, Classification.**

I. Introduction

Recent advances in science and technology have transformed the software industry, increasing the demand for reliable software systems, especially for complex calculations and online functionality. Everyday software, including web applications, is a critical component of economic growth and is characterized by continuous evolution and increasing complexity [1]. Development and testing are vital stages in the software development lifecycle (SDLC) [3], requiring effective validation techniques to ensure reliability, security, and reusability. The complexity of modern software and web applications often involving intricate designs and sensitive data poses significant challenges in testing, with potential gaps leading to serious issues such as data breaches and fraud [34]. Web systems, essential for e-commerce and banking, must undergo rigorous security testing to identify vulnerabilities [6]. Despite advancements in development, testing remains costly and time-consuming, accounting for over 50% of total software expenditures [8]. The testing process carries risks of human error, and evaluating correctness can be complex, especially in systems like banking software with intricate functionalities [4]. Additionally, various input interfaces complicate vulnerability detection, necessitating the use of automated strategies for security testing to effectively cover diverse user roles and inputs [5].

Metamorphic Testing (MT) is a technique that has proven useful in certain situations to tackle the challenge of the oracle problem [10]. The principle behind MT [9] is that it may be easier to analyze the relationships between the results of multiple test executions, referred to as Metamorphic Relations (MRs), rather than specifying the input-output behavior of a system [9]. MT utilizes MRs to determine system properties by automatically transforming the initial test input into a subsequent test input [9]. If the system fails to comply with the MRs when tested with both the initial and subsequent inputs, it is inferred that it is defective [9]. A considerable amount of research has focused on developing MT methods for specific domains such as computer graphics, web services, and embedded systems [12]–[15], and the corresponding MRs are essential for enhancing the efficiency of fault detection in testing. Traditionally, pinpointing these relations involves a labor-intensive, error-prone manual process, often requiring input from domain experts, mainly when dealing with intricate programs. Hence, developing automated techniques to identify MRs is a promising avenue. Such automation has the potential to significantly improve the efficiency and effectiveness of MT, rendering it a more feasible and dependable method for guaranteeing the reliability of any system. Therefore, we wanted to use MT in the domain of online banking applications to investigate its effectiveness in security vulnerability testing.

Our research goal is described as follows:

- **RG**: *To apply MT to tackle the Test Oracle problem in security vulnerability testing of online banking applications.*

We systematically define MRs that capture properties (i.e., characteristics that are compromised when the system is at risk) among the top 10 OWASP vulnerabilities found in online banking applications and automate testing using these MRs for such vulnerable-prone programs. These MRs serve as guidelines or rules governing how the program should change in response to different inputs or variations. They are crafted to capture specific program characteristics that are particularly vulnerable or compromised when the system is at risk. An example of an MR to identify *bypass authorization schema* vulnerabilities is *a web system that should return different responses for two users when the first user requests a URL provided by the GUI* (e.g., in HTML links). In other words, a user should not be able to access URLs that are not provided by the GUI directly. We have developed a technique to automate the testing process by predicting these MRs. Automating the testing process reduces the manual effort and potential errors associated with traditional testing methods.

Here, we present an automated metamorphic testing approach that helps test engineers specify metamorphic relations to capture security requirements of Web systems (i.e., online banking) and automatically detect vulnerabilities (i.e., violations of security requirements) by predicting metamorphic relations. Our approach is constructed on the following novel contributions:

- A catalog of 8 MRs targeting the well-known top 10 OWASP's security vulnerabilities commonly found in online baking applications, and
- A framework that automatically predicts MRs for unseen programs, that are prone to vulnerabilities, using a Graph-based Convolutional Neural Network (GCNN) model.

We applied our approach to 679 vulnerable-prone programs from different sources. The approach automatically detected MRs with prediction scores greater than 80% for most MRs. Considering these results and assessing the effort involved, our approach is practical and effective in addressing the Oracle issue of automatically testing online banking applications.

This paper is organized in the following manner. Section II offers background information on MT and details the testing process for baking applications. In Section III, we outline our approach. Section IV explains the fundamental experimental setup. Section V shares the paper's findings. Finally, Section VI addresses the threats to validity, leading to the conclusion in Section VII.

II. Background

This section introduces MT and its operation according to the corresponding MRs. It also illustrates the challenges of testing vulnerabilities for banking applications and how MT can alleviate the issue.

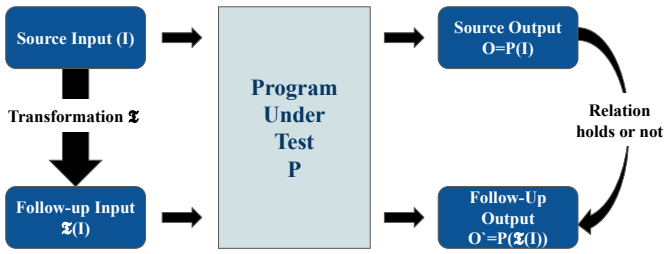

Fig. 1. Overview of the Metamorphic Testing (MT) Process

A. Metamorphic Testing

Metamorphic Testing (MT) is a technique designed to address the well-known Test Oracle problem [10]. This problem arises when it is difficult or impossible to determine the correctness of individual program output. MT, developed by Chen et al. [9], provides a solution by checking if a program satisfies specific predefined properties called Metamorphic Relations (MRs). These MRs describe how changes to a program's input should affect its output [9]. If the output does not behave as expected according to these MRs, it may indicate a fault in the program [9]. MT is particularly valuable for identifying defects in programs where traditional Test Oracles are unavailable or impractical [10]. By examining the relationships between inputs and outputs across multiple executions, MT can detect faults even when the correct result of each execution is unknown [11].

The general steps for implementing MT are as follows (Figure 1) [9]:

- **Identify MRs:** Define a set of MRs that the program under test should satisfy.
- **Create Initial Test Cases:** Develop a set of test cases to serve as the source inputs.
- **Generate Follow-Up Test Cases:** Apply input transformations specified by the MRs to create follow-up test cases from the initial ones.
- **Execute and Compare Outputs:** Run both the initial and follow-up test cases and check if the output changes align with the expected behavior described by the MRs. If a runtime violation of an MR occurs, it indicates a fault in the program.

A simple and widely used example of MT is testing the SINE function $y = \sin(x)$. According to its property, for any input angle x, adding 2π to the input should not change the output. This means $y = \sin(x) = \sin(x + 2\pi)$, making it a valid MR. Using this property, the function can be tested as follows:

- Create a source test case with input x and output $\sin(x)$.
- Generate a follow-up test case by applying the transformation $x' = x + 2\pi$ and calculate $y = \sin(x') = \sin(x + 2\pi)$.

- Compare the outputs. If $sin(x') \neq sin(x)$, the MR is violated, indicating a fault in the sine function's implementation.

By systematically applying these steps, MT provides an effective and efficient alternative to detect faults in the absence of traditional Test Oracles [10].

B. Testing Banking Applications

Banking applications handle sensitive financial and personal data, making their security critical. Vulnerabilities, such as weak authentication, injection flaws, insecure communication, and business logic errors, can lead to severe consequences, including financial fraud and reputational damage [5]. These vulnerabilities often arise from the complexity of banking systems, which include intricate workflows, multiple user roles, and diverse input interfaces such as web pages, forms, and cookies [5]. Common vulnerabilities in banking applications include authentication and authorization issues, injection attacks, cross-site scripting (XSS), business logic flaws, insecure communication, and third-party dependencies [5]. Several testing techniques are commonly employed to detect vulnerabilities in banking applications, including the use of static application security testing (SAST) [17], dynamic application security testing (DAST) [18], penetration testing, fuzz testing, risk-based testing [5], and manual code reviews [5]. These methods often face challenges, such as restricted test coverage, proneness to human error, and difficulties in addressing dynamic behaviors, commonly called the Oracle Problem [16]. Frequent configuration changes and diverse input parameters in banking applications further complicate the testing process.

MT addresses these gaps by leveraging MRs. Unlike traditional methods, MT does not require precise expected results, making it particularly useful in scenarios where Test Oracles are unavailable or impractical [10]. MT resolves the Oracle Problem by focusing on input-output relationships. Additionally, it automates test case generation and validation, reducing human error and systematically covering complex input combinations to improve test coverage [10]. For instance, SQL injection is a common vulnerability in banking applications. Attackers can manipulate input fields, such as login forms or search bars, to execute unauthorized SQL queries. If user inputs are not properly validated, an attacker could bypass authentication by submitting crafted inputs and gaining unauthorized access to sensitive user data. This vulnerability allows attackers to manipulate or delete records, steal financial information, and perform fraudulent transactions. MT can verify the system's behavior by testing transformed inputs to ensure proper handling and validation, mitigating such risks. As banking systems grow in complexity, MT offers a scalable and practical approach to ensure their security and reliability. By addressing the limitations of traditional testing methods, MT enhances the efficiency and effectiveness of vulnerability detection in banking applications.

III. Approach

In this section, we introduce a new approach to automating the MT technique. We predict MRs for vulnerable-prone programs commonly found in banking applications, which fall under OWASP's Top 10 vulnerabilities. The proposed method, illustrated in Figure 2, consists of three key steps:

1) identifying MRs for vulnerabilities related to banking applications among OWASP's Top 10 vulnerabilities, making our method directly applicable to real-world scenarios,
2) generating control flow graphs that capture the execution behavior of vulnerable-prone Java programs, and
3) applying a graph-based model to analyze CFG datasets for MR prediction, ensuring a comprehensive and accurate assessment.

In the first step, we define MRs based on the characteristics and behaviors of vulnerabilities that can be tested using MT in Java-based banking applications. In the second step, we construct CFG representations from Java source code to model program execution flows. The generated CFGs provide a structured, graphical representation of program behavior, essential for further analysis. In the final step, we employ graph-based classification models to analyze the CFGs. Specifically, we introduce a graph convolutional neural network to automatically learn predictive patterns from CFG data, enabling efficient and accurate vulnerability testing in banking applications by predicting MRs.

A. Metamorphic Relations (step 1)

The first task is to identify MRs. We developed 8 MRs from the OWASP's Top 10 (2021) list. To identify the MRs, we focused on properties related to vulnerabilities found in online banking web applications. Among the top 10, we identified MRs for only 8, as these are the most common. We named the MRs by numbering them from 1 to 8 (e.g., MR1, MR2, etc.), and we retained the numbering (e.g., A01, A02, etc.) from the original list in case of vulnerabilities. The descriptions for each MR are as follows:

- **MR 1 (A01: Broken Access Control: Authorization-Based Access Control):** Let U_a represent an authorized user and U_u represent an unauthorized user. Let R denote a restricted resource. If $f(U_a, R) \longrightarrow access\_granted$, then $f(U_u, R) \longrightarrow access\_denied$.
- **MR 2 (A02: Cryptographic Failures: HTTPS vs. HTTP Transmission):** Let P represent plaintext data (e.g., credentials or transaction details) and T denote the transmission protocol. If $T(HTTPS, P) \longrightarrow encrypted$, then $T(HTTP, P) \neq encrypted$.
- **MR 3 (A03: SQL Injection: Authentication Bypass):** Let I_v represent a valid login input and I_m denote a malicious SQL payload. If $DB\_query(I_v) \longrightarrow valid\_response$, then $DB\_query(I_m) \neq valid\_response$.
- **MR 4 (A04: Insecure Design: Password Reset Token Exposure):** Let T_s represent a securely generated token, and T_u represent an unsecured token. If

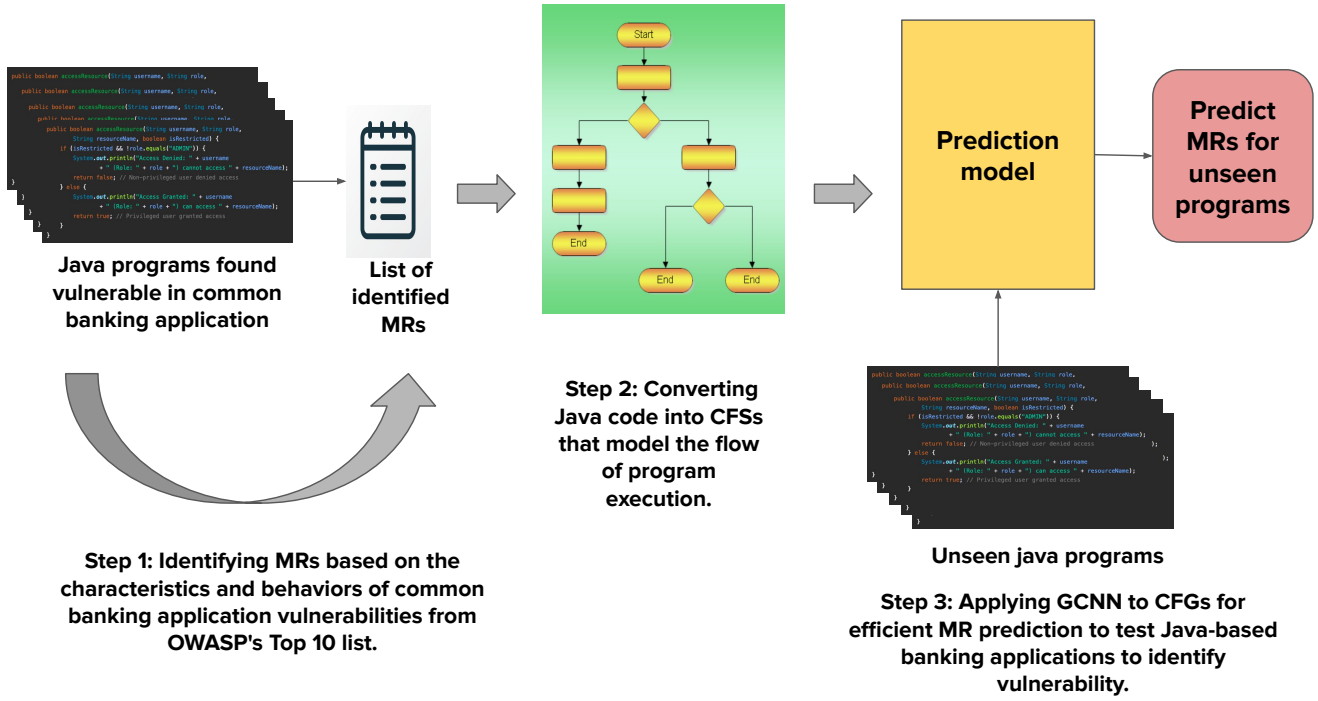

Fig. 2. Overview of the proposed approach for automating metamorphic testing (MT) to predict MRs for vulnerabilities in banking applications. The method consists of three main steps: (1) identifying metamorphic relations (MRs) for vulnerabilities from OWASP's Top 10 list related to banking applications, (2) generating control flow graphs (CFGs) from Java source code to capture execution behavior, and (3) applying a graph convolutional neural network (GCNN) to analyze the CFGs for MR prediction.

$Validate(T_s) \longrightarrow valid\_reset$, then $Validate(T_u) \neq valid\_reset$.

- **MR 5 (A05: Security Misconfiguration: Default Credentials Exposure):** Let C_u represent a user-defined credential and C_d denote a default credential. If $Auth(C_u) \longrightarrow secure\_login$, then $Auth(C_d) \neq secure\_login$.
- **MR 6 (A07: Authentication Failures: Session Expire):** Let S_v represent a session within its valid time frame, and S_e denote an expired session. If $Auth(S_v) \longrightarrow access\_granted$, then $Auth(S_e) \longrightarrow access\_denied$.
- **MR 7 (A08: Software and Data Integrity Failures: Checked vs. Unchecked Transactions):** Let T_c denote a transaction that includes checksum validation, while T_u represents a transaction that does not include checksum validation. If $Process(T_c) \longrightarrow valid\_transaction$, then $Process(T_u) \neq valid\_transaction$.
- **MR 8 (A10: Server-Side Request Forgery (SSRF): Allowlisted vs. Denylisted URLs):** Let U_a represent an allowlisted URL and U_d represent a denylisted URL. If $Request(U_a) \longrightarrow allowed$, then $Request(U_d) \longrightarrow blocked$.

B. Function Representation using Control Flow Graphs (CFG) (step 2)

The next step in this approach involves converting a function into its Control Flow Graph (CFG). This representation is chosen because it facilitates the extraction of information regarding the sequence of operations within a control flow path, which directly corresponds to the MRs satisfied by a given function [20].

A CFG is a directed graph, $G_f = (V, E)$, representing a function *f*. Each node $v_x \in V$ corresponds to a statement *x* in *f*, with the operation performed in *x* labeled as *label*(v_x). If *x* and *y* are statements in *f*, and *y* executes immediately after *x*, an edge $e = (v_x, v_y) \in E$ is established. The control flow of *f* is defined by all edges in the graph, while the entry and exit points are represented by nodes v_{start} and v_{exit}, respectively [20].

To construct the CFGs, we utilize the *Soot*[1] framework. This tool generates CFGs in *Jimple*, a typed three-address intermediate representation of Java code, where each CFG node represents an atomic operation [21]. After generating the CFGs, we refine them by labeling each node according to the operation it performs. Additionally, we annotate all method call nodes with their return types.

[1]https://www.sable.mcgill.ca/soot/

This study aims to develop a technique for identifying MRs from vulnerable characteristics in Java source code. In this context, CFGs produced as intermediate representations of compiled source code serve as inputs for prediction models to detect MRs to test vulnerable programs. Prior research has demonstrated the successful application of CFGs in various fields, including malware analysis [22], [23] and software plagiarism detection [24], [25]. Since semantic errors often become apparent only at runtime, analyzing execution flows can be valuable in distinguishing faulty patterns from correct ones.

C. Prediction Model (step 3)

We used a graph model to predict MRs for vulnerable-prone programs. The Graph Convolutional Network (GCN) is a dynamic graphical model that processes large-scale graphs with intricate structural relationships [26]. Unlike simple node-based representations, a vertex in the GCNN framework is not merely a token but it encapsulates a rich set of features derived from its connectivity within the graph [26]. For instance, in a CFG context, each vertex signifies an operation that may consist of multiple attributes, including the instruction type and operands. In the GCNN model, the first layer is called the embedding layer [26], where each vertex is mapped to a real-valued vector corresponding to its feature representation. Next, we apply two graph convolutional layers [26] that iteratively update the node embeddings by aggregating information from neighboring nodes. These layers capture local graph structures and enhance feature representations at multiple levels. The network has batch normalization layers to stabilize training and improve convergence. A ReLU activation function [26] is applied after each convolutional operation to introduce non-linearity. After the convolutional layers, a global mean pooling layer is used to extract a unified feature representation of the entire graph. Unlike standard CNNs with fixed input dimensions, graphs feature varying sizes, resulting in inconsistencies in feature extraction [26]. Pooling addresses this by consolidating the learned node features into a single vector representation. Ultimately, the feature vector is processed through a fully connected layer and an output layer, where categorical distributions for classification tasks are generated. In this manner, a graph-based model automates vulnerability testing in Java programs by predicting MRs and capturing execution flow characteristics.

IV. Experiments

We experimented with a dataset of programs prone to vulnerabilities commonly found in banking applications. Our aim was to generate a graph-based model to predict MRs for those programs. This section provides a detailed description of the dataset, the experimental setup, and the evaluation metrics used.

A. Dataset

We constructed a dataset of 679 Java programs prone to vulnerabilities in online banking applications. This dataset is curated based on the OWASP Top 10 vulnerabilities, which represent the most critical security issues for web applications. We built the dataset by collecting Java programs from the following sources:

- **AI-generated vulnerable programs:** Similar code samples were generated using ChatGPT (265) and Meta AI LLaMA (265), focusing on the OWASP Top 10 vulnerabilities.
- **Open-source repositories:** Vulnerable Java applications sourced from open-source projects on GitHub (98).
- **Manually curated examples:** Additional vulnerable samples are chosen to ensure diversity (51).

Each program contains security flaws such as SQL injection, broken access control, authentication failures, and other OWASP vulnerabilities. Figure 3 is an example of a vulnerable-prone Java program generated from ChatGPT. Once the Java programs are collected, they are converted into CFGs representing the program's execution flow. The CFGs are generated using static analysis tools called Soot (a Java optimization framework). It extracts control flow structures from Java bytecode. The generated CFGs represent nodes as instructions or basic blocks, while edges indicate execution flow between these instructions. Each CFG is stored in DOT file format [27], which provides a graphical representation of program execution.

B. Experimental Setup

The experiment aimed to analyze and predict MRs for vulnerable-prone Java programs by leveraging CFGs and prediction models. Each node in the CFG is assigned a label based on its operation (e.g., ASSIGNMENT, IF_CONDITIONS). These labels were converted into unique 16-dimensional binary vectors, which serve as node features in the graph representation. For example, all the nodes with the same type of operations have the same 16-dimensional binary vectors. Graph nodes were assigned unique features based on their operations, and edges were structured as adjacency matrices. The dataset was split into training and test sets to evaluate the model's performance. These splits were done based on three categories. They were:

1) train with ChatGPT, Open-source repositories, and manually crafted data. Test with Meta AILLaMa data,
2) train with Meta AILLaMa, Open-source repositories, and manually crafted data. Test with ChatGPT data, and
3) train with Meta AILLaMa and ChatGPT data. Test with Open-source repositories and manually crafted data.

Also, binary class labels were supplied to train the models for each MR.

To ensure a comprehensive evaluation, we compare the effectiveness of Graph Neural Networks (GNNs) against traditional prediction models. A two-layer GCNN model followed by a fully connected layer is used to predict vulnerability classes. We used Support Vector Machine (SVM) approaches with two feature extraction types developed in previous studies [28], [29]. One is random kernel-based SVM [28] and SVM

```java
public boolean accessResource(String username, String role,
        String resourceName, boolean isRestricted) {
    if (isRestricted && !role.equals("ADMIN")) {
        System.out.println("Access Denied: " + username
                + " (Role: " + role + ") cannot access " + resourceName);
        return false; // Non-privileged user denied access
    } else {
        System.out.println("Access Granted: " + username
                + " (Role: " + role + ") can access " + resourceName);
        return true; // Privileged user granted access
    }
}
```

Fig. 3. Simplified Access Control Method Implementing Basic Role-Based Authorization. This method checks if a user with a specific role is allowed to access a restricted resource. Only users with the "ADMIN" role are granted access to restricted resources, while others are denied. The output clearly indicates whether access is granted or denied based on the user's role. Metamorphic Relation 1 (MR1) for Authorization-Based Access Control (RBAC) Violation can be used to test this method, ensuring that privileged users are granted access while non-privileged users are correctly denied.

with Bag-of-Words (BoW) [29]. We also used the multi-layer perceptron (MLP), which is a fully connected feedforward neural network trained on the same dataset [30]. The results show the average prediction scores for all the categories of train tests split together.

C. Evaluation Measures

The approaches are evaluated using two widely recognized performance metrics: the area under the receiver operating characteristic (ROC) curve (AUC) and accuracy.

The AUC (Area Under the Curve) provides a more comprehensive evaluation of the model's performance [31]. It assesses the classifier's ability to distinguish between two classes by quantifying the area under the ROC curve, which plots the true positive rate (TPR) against the false positive rate (FPR) at different classification thresholds [31]. The comprehensive nature of AUC evaluation provides reassurance and confidence in the model's performance, especially when class distributions are imbalanced [31]. A classifier with an AUC of 1.0 perfectly distinguishes between the two classes, while an AUC of 0.5 indicates no better performance than random guessing [31].

Accuracy measures the proportion of correctly classified instances among all predictions and is a fundamental metric for evaluating classification models [32]. Accuracy, while a fundamental metric for evaluating classification models, can be misleading when class imbalances exist. In such cases, where one class has significantly fewer samples than the other, a model biased toward the majority class may achieve high accuracy but perform poorly in detecting minority-class instances. [32].

V. Results And Discussion

The performance of four classification models—Support Vector Machine (SVM) with Random Walk kernel, SVM with Bag of Words (BoW), Multilayer Perceptron (MLP), and Graph Convolutional Neural Network (GCNN) was assessed using two key performance metrics, the Area Under the Receiver Operating Characteristic Curve (AUC) and accuracy.

Fig. 4. AUC Score Distribution for Metamorphic Relations (MR1–MR8) Using SVM-Random Walk, SVM-BoW, MLP, and GCNN Models.

Fig. 5. Accuracy Score Distribution for Metamorphic Relations (MR1–MR8) Using SVM-Random Walk, SVM-BoW, MLP, and GCNN Models.

This evaluation was conducted to determine the effectiveness of the models in predicting MRs for vulnerable-prone Java programs in banking applications.

The GCNN consistently outperformed the other models, achieving the highest AUC scores across most MRs. Figure 4 shows GCNN achieved AUC scores of 0.95 (MR1), 0.92 (MR3), 0.92 (MR5), and 0.94 (MR7). It indicates its capability to capture complex structural patterns from the CFGs of the programs. The SVM with Random Walk kernel also demonstrated strong performance, particularly for MR2 (0.83) and MR3 (0.85). In contrast, the MLP model showed relatively lower AUC scores. Here, the performance declined notably for MR5 (0.6) and MR3 (0.66). It illustrates its limitations in addressing structural graph data compared to GCNN. SVM-BoW showed average performance, with scores ranging from 0.7 to 0.81. It reflects its dependency on text-based feature representation, which may not fully capture the complex program behaviors.

The accuracy results further confirm the AUC findings, emphasizing the robustness of GCNN in predictive performance. Figure 5 shows that GCNN achieved the highest accuracy for

MR6 (0.91), MR3 (0.89), and MR8 (0.87), demonstrating its robustness in accurately classifying MRs. SVM with Random Walk kernel upheld competitive accuracy, particularly in MR4 (0.89) and MR5 (0.83). However, its performance slightly varied across different MRs. The MLP model lagged, with the lowest MR2 (0.62) and MR6 (0.63) accuracy, reflecting its struggles with feature extraction from complex graph data. SVM-BoW performed relatively well. It achieved accuracy scores of 0.81 (MR7) and 0.8 (MR8) but did not match the consistency observed with GCNN.

The results emphasize the advantage of leveraging graph-based models, particularly GCNN, in analyzing control flow graphs for MR prediction. The AUC and accuracy scores of GCNN can be attributed to its ability to learn hierarchical feature representations from graph-structured data, which traditional models like SVM and MLP cannot fully manage. The relatively strong performance of SVM with Random Walk kernel suggests that kernel-based approaches still hold value, especially when dealing with graph structure data. However, the lower performance of MLP highlights the challenges neural networks face, such as the lack of specialized architectures for graph data. Overall, the findings validate the advantage of leveraging graph-based models for MR prediction for vulnerable-prone Java programs, particularly in domains where program structure plays a crucial role.

VI. Threats to validity

External validity [33] is concerned with the generalizability of the findings beyond the specific dataset and experimental setup used in this study. The study was conducted on a dataset that does not represent all real-world applications. The results may not generalize to datasets with different characteristics, such as varying feature distributions, noise levels, or larger sample sizes. While the models perform well on the given dataset, their scalability to large-scale real-world applications remains uncertain. The effectiveness of GCNNs depends on the graph structure of the data. If the dataset does not naturally fit into a graph representation, the performance of GCNNs might be inferior to traditional models like SVM or MLP. This limits the generalization of GCNNs to various classification tasks.

Internal validity [33], which refers to potential biases, errors, or confounding factors affecting the experimental results, is critical. The study relies on widely used frameworks such as Soot, Scikit-learn, TensorFlow, and Graph Neural Network (GNN) libraries like PyTorch-Geometric. However, while popular, these frameworks may still contain undocumented bugs or implementation inconsistencies that could impact the experimental outcomes. Therefore, further validation on more extensive and diverse datasets is necessary to confirm the robustness of the findings and to ensure the ongoing relevance of this research.

VII. Conclusion

Our research goal was *To apply MT to tackle the Test Oracle problem in security vulnerability testing of online banking applications.*

We evaluate the effectiveness of automatically predicting MRs using graph models for vulnerable-prone Java programs commonly found in banking applications. MT is a testing technique for programs lacking a suitable Test Oracle. Manually identifying such MRs for various applications poses challenges for testers. In prior work, we developed a graph kernel-based machine learning approach for predicting MRs using supervised learning and control flow graph features tailored for complex scientific programs. Results indicate that utilizing control flow graph features to compute the graph kernel of a testing program for training a machine learning model enhances MR prediction. This research evaluates the effectiveness of using graph representations directly to train graph models of vulnerable-prone Java programs. Graph Convolutional Neural Networks (GCNNs) are evaluated using a control flow graph of vulnerable-prone programs in online banking applications. Eight types of metamorphic relations are manually identified for predicting MRs that cover the properties of vulnerabilities among the top 10 OWASP vulnerabilities. These MRs are predicted on Graph Convolutional Neural Networks (GCNNs), MLP, random kernel-based SVM, and SVM with bag-of-words. The result shows the advantage of leveraging graph-based models, particularly GCNN, in analyzing control flow graphs for MR prediction. GCNN consistently shows the highest AUC scores across most MRs, with values ranging from 0.76 to 0.95, indicating its ability to discriminate between classes.

The proposed method has several potential extensions. Firstly, to enhance external validity, utilizing a larger dataset with various functions can improve accuracy. Currently, the method focuses on predicting a single metamorphic relation, relying on a binary classification system to train predictive models. However, it can be adapted to predict metamorphic relations using multi-class models. Additionally, this approach could be expanded to include datasets from other programming languages, such as Python, C, and Fortran, which would be advantageous for the scientific community.

References

[1] M. M. Lehman and L. A. Belady, *Program Evolution: Processes of Software Change.* USA: Academic Press Professional, Inc., 1985.

[2] Type, P-type, S-type systems. E. (n.d.). Retrieved May 2, 2023, from https://denrox.com/post/e-type-p-type-s-type-systems.

[3] R. Sanders and D. Kelly, "The challenge of testing scientific software," *in Proceedings of the 2008 Workshop on Software Engineering for Computational Science and Engineering*, Leipzig, Germany, May 2008, pp. 1–7.

[4] S., Alireza, G., Gregory and M., Ehsan. (2022). Mapping the Structure and Evolution of Software Testing Research Over the Last Three Decades. Journal of Systems and Software. 10.1016/j.jss.2022.111518.

[5] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Security testing: A survey," *Advances in Computers*, vol. 101, pp. 1–51, 2016, doi: 10.1016/bs.adcom.2015.11.003.

[6] C. Haley, R. Laney, J. Moffett, and B. Nuseibeh, "Security requirements engineering: A framework for representation and analysis," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 133–153, Jan. 2008, doi: 10.1109/TSE.2007.70754.

[7] M. Staats, M. W. Whalen, and M. P. Heimdahl, “Programs, tests, and oracles: The foundations of testing revisited,” *in Proceedings of the 33rd International Conference on Software Engineering (ICSE’11)*, Waikiki, Honolulu, HI, USA, May 2011, pp. 391–400, doi: 10.1145/1985793.1985849.

[8] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015, doi: 10.1109/TSE.2014.2372785.

[9] T. Y. Chen, S.-C. Cheung, and S.-M. Yiu, “Metamorphic testing: A new approach for generating next test cases,” *The Hong Kong University of Science and Technology*, Tech. Rep., 1998.

[10] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, “How effectively does metamorphic testing alleviate the oracle problem?” *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 4–22, Jan. 2014, doi: 10.1109/TSE.2013.47.

[11] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, “A survey on metamorphic testing,” *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, Sep. 2016, doi: 10.1109/TSE.2016.2532875.

[12] R. Guderlei and J. Mayer, “Towards automatic testing of imaging software by means of random and metamorphic testing,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 17, no. 6, pp. 757–781, 2007, doi: 10.1142/S0218194007003540.

[13] T. Y. Chen, F.-C. Kuo, W. Ma, W. Susilo, D. Towey, J. Voas, and Z. Q. Zhou, “Metamorphic testing for cybersecurity,” *Computer*, vol. 49, no. 6, pp. 48–55, Jun. 2016, doi: 10.1109/MC.2016.166.

[14] K. Rahman and C. Izurieta, ”An Approach to Testing Banking Software Using Metamorphic Relations,” *2023 IEEE 24th International Conference on Information Reuse and Integration for Data Science (IRI)*, Bellevue, WA, USA, 2023, pp. 173-178, doi: 10.1109/IRI58017.2023.00036.

[15] F.-C. Kuo, T. Y. Chen, and W. K. Tam, “Testing embedded software by metamorphic testing: A wireless metering system case study,” *in Proceedings of the 36th IEEE Conference on Local Computer Networks (LCN’11)*, Bonn, Germany, Oct. 2011, pp. 291–294, doi: 10.1109/LCN.2011.6115199.

[16] K. Rahman and C. Izurieta, ”A Mapping Study of Security Vulnerability Detection Approaches for Web Applications,” *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Gran Canaria, Spain, 2022, pp. 491-494, doi: 10.1109/SEAA56994.2022.00081.

[17] Z. Wadhams, C. Izurieta, and A. M. Reinhold, “Barriers to using static application security testing (SAST) tools: A literature review,” *in Proceedings of the Workshop on Human-Centric Software Engineering & Cyber Security (HCSE&CS 2024)*, Sacramento, CA, USA, Nov. 2024.

[18] V. Somi, “A comparative analysis and benchmarking of dynamic application security testing (DAST) tools,” *Journal of Engineering and Applied Sciences Technology*, vol. 1, no. 6, pp. 1–6, 2024, doi: 10.47363/JEAST/2024(6)E139.

[19] P. Godefroid, M. Y. Levin, and D. A. Molnar, “Automated whitebox fuzz testing,” *in Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2008, vol. 8.

[20] F. E. Allen, “Control flow analysis,” *in Proceedings of a Symposium on Compiler Optimization*, New York, NY, USA, 1970, pp. 1–19, doi: 10.1145/800028.808479.

[21] R. Vallee-Rai and L. J. Hendren, “Jimple: Simplifying Java bytecode for analyses and transformations,” *Tech. Rep.*, 1998.

[22] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, ”Graph-based malware detection using dynamic analysis”, *Journal in computer virology*, 7(4):247–258, 2011.

[23] D. Bruschi, L. Martignoni, and M. Monga, ”Detecting selfmutating malware using control-flow graph matching,” *In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 129–143. Springer, 2006.

[24] D.K. Chae, J. Ha, S. W. Kim, B. Kang, and E. G. Im, ”Software plagiarism detection: a graph-based approach,” *In Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 1577–1580. ACM, 2013.

[25] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, ”Detecting code reuse in android applications using component-based control flow graph,” *In IFIP International Information Security Conference*, pages 142–155. Springer, 2014.

[26] T. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *in Proceedings of the International Conference on Learning Representations (ICLR’17)*, Toulon, France, Apr. 2017.

[27] E. R. Gansner and S. C. North, “An open graph visualization system and its applications to software engineering,” *Software: Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 2000, doi: 10.1002/1097-024X(200009)30:11¡1203::AID-SPE338 3.0.CO;2-N.

[28] K. Rahman and U. Kanewala, “Predicting metamorphic relations for matrix calculation programs,” *in Proceedings of the 3rd International Workshop on Metamorphic Testing (MET ’18)*, Gothenburg, Sweden, May 2018, pp. 10–13, doi: 10.1145/3193977.3193983.

[29] K. Rahman, I. Kahanda, and U. Kanewala, “MRpredT: Using text mining for metamorphic relation prediction,” *in Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW’20)*, Seoul, South Korea, Jun. 2020, pp. 420–424, doi: 10.1145/3387940.3392250.

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, 1986, doi: 10.1038/323533a0.

[31] T. Fawcett, “An introduction to ROC analysis,” *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, Jun. 2006, doi: 10.1016/j.patrec.2005.10.010.

[32] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, “Supervised machine learning: A review of classification techniques,” *Artificial Intelligence Review*, vol. 26, no. 3, pp. 159–190, Nov. 2007, doi: 10.1007/s10462-007-9052-3.

[33] W. R. Shadish, T. D. Cook, and D. T. Campbell, ”Experimental and Quasi-Experimental Designs for Generalized Causal Inference,” Boston, MA, USA: Houghton Mifflin, 2002.

[34] C. Izurieta and M. Prouty, ”Leveraging SecDevOps to Tackle the Technical Debt Associated with Cybersecurity Attack Tactics,” *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, Montreal, QC, Canada, 2019, pp. 33-37, doi: 10.1109/TechDebt.2019.00012.