A Hybridized Approach for Testing Neural Network Based Intrusion Detection Systems

Faqeer ur Rehman
Gianforte School of Computing
Montana State University
Bozeman, MT, USA
faqeer.rehman@student.montana.edu

Dr. Clemente Izurieta
Gianforte School of Computing
Idaho National Laboratories
Montana State University
Bozeman, MT, USA
clemente.izurieta@montana.edu

***Abstract*—Enhancing the trust of machine learning-based classifiers with large input spaces is a desirable goal; however, due to high labeling costs and limited resources, this is a challenging task. One solution is to use test input prioritization techniques that aim to identify and label only the most effective test instances. These prioritized test inputs can then be used with some popular testing techniques e.g., Metamorphic testing (MT) to test and uncover implementation bugs in computationally complex machine learning classifiers that suffer from the oracle problem. However, there are certain limitations involved with this approach, (i) using a small number of prioritized test inputs may not be enough to check the program correctness over a large variety of input scenarios, and (ii) traditional MT approaches become infeasible when the programs under test exhibit a non-deterministic behavior during training e.g., Neural Network-based classifiers. Therefore, instead of using MT for testing purposes, we propose a metamorphic relation to solve a data generation/labeling problem; that is, enhancing the test inputs effectiveness by extending the prioritized test set with new tests without incurring additional labeling costs. Further, we leverage the prioritized test inputs (both source and follow-up data sets) and propose a statistical hypothesis testing (for detection) and machine learning-based approach (for prediction) of faulty behavior in two other machine learning classifiers (Neural Network-based Intrusion Detection Systems). In our case, the problem is interesting in the sense that injected bugs represent the high accuracy producing mutated program versions that may be difficult to detect by a software developer. The results indicate that (i) the proposed statistical hypothesis testing is able to identify the induced buggy behavior, and (ii) Random Forest outperforms and achieves the best performance over SVM and k-NN algorithms.**

***Index Terms*—Machine learning, Machine learning testing, Statistical hypothesis testing, Neural networks, Prioritized inputs, Metamorphic testing, Intrusion detection systems**

I. Introduction

Machine Learning (ML) provides core functionality to many critical application domains, such as bioinformatics, network security, collaborative robots, and self-driving vehicles. Developing high-quality ML models to solve real-world classification, prediction, and clustering problems is always desirable because a small bug in these critical high-consequence applications can lead to disastrous consequences and can pose serious threats to life and property [10] [11] [12]. Therefore, it is very important to verify their correctness before deploying them in a production environment. However, the computational complexity, large input space, and lack of a test oracle raise serious challenges in order to verify their correct functionality. Consequently, these applications end up in a category of non-testable programs, also known as programs suffering from the *Oracle Problem* [3].

From a quality assurance perspective, it is always desirable to verify the correctness of ML-based classifiers over a large range of test scenarios. However, the high cost of obtaining a test oracle, a.k.a. labeling a large number of test instances, makes it a resource-intensive and infeasible task. One of the solutions proposed in the literature is to use test input prioritization techniques that aim to identify and label only the most effective test instances [16] [17]. However, concerns like, whether the small number of prioritized test inputs are enough to test the correctness of critical ML applications over a diverse set of data, are unaddressed. We propose a *Metamorphic Relation* (MR) that aims to target this data collection/labeling problem by extending the prioritized test set with new tests without incurring additional labeling costs. It thus allows the software tester to check the program correctness over a diverse set of input scenarios.

Metamorphic Testing (MT) is considered an effective testing technique to alleviate the oracle problem in testing ML applications [4] [6] [7] [15]. MT is not only a testing technique but can also be used to generate domain specific data using valid transformations to the input data e.g., rotation, scaling, reflection etc. At the heart of MT are MRs that are derived from the necessary characteristics of the program under test. Each MR is composed of a source test case and a follow-up test case. A valid transformation performed on the source test-case generates a new test case, known as a follow-up test case. Instead of verifying the output for individual inputs, the relation between the input and its associated output is used to verify the program correctness. The MR is said to be violated if the result of the source and follow-up test case does not adhere to the relation specified in the corresponding MR. A violation of a MR will indicate that there is some potential bug in the system.

One of the challenges faced in applying tradition MT ap-

978-1-6654-3545-1/21/$31.00 ©2021 IEEE

proach to test ML applications is when the program under test exhibits a stochastic behavior in training e.g., Artificial Neural Network (ANN) based classifiers (due to random initialization of weights). For example, given the same inputs, the NN-based application may produce slightly different output in each run; thus rendering traditional MT techniques inadequate if they rely on using the equality operator to check the relation specified in the MR. One of the solutions proposed in the literature (for getting deterministic results) is either to fix the random seed(s) or initialize the network weights with the same constant values [6] [7]. However, the proposed solution has its own limitations i.e., it is only applicable in an environment where either the random seeds or the weights can be fixed, which is not always possible. Furthermore, even if somehow the random seeds have successfully been fixed, it is not guaranteed that we can obtain deterministic results on GPUs because of inherent non-determinism introduced by NVidia CUDA libraries [6]. Also, in practice, it may not be viable for a software tester to identify and then fix the seeds in a highly complex system internally using a large number of third-party libraries, especially when some of the libraries do not provide any option at all. To alleviate this problem, our prior work focuses on proposing statistical measures for testing deep neural network based applications in a non-deterministic environment [9].

The above-highlighted limitations motivate us to propose an approach that can not only be used to extend the prioritized test set with new test inputs (using the proposed MR) but can also leverage these inputs (both source and follow-up test sets) to test NN-based classifiers (in both deterministic and non-deterministic environments) using a combination of statistical and ML techniques. The purpose of performing statistical analysis is to check whether the difference between the results produced by original and mutated program versions is statistically significant. If so, the buggy behavior is said to be detected. Next, we use this knowledge to build ML classifier that can be used to predict whether, for the given prioritized test inputs, the output (probability distribution over classes) produced by an NN-based classifiers under test exhibits a 'buggy' or 'non-buggy' behavior. These prioritized test inputs can be a part of permanent test case repository that the organization frequently uses to test the new release in regression testing environment and reducing the overall cost of software testing phase.

To show the effectiveness of the proposed approach, we have worked with detecting and predicting buggy behavior in two NN-based Intrusion Detection Systems (N-IDSs). The following are the main contributions made in this paper:

- Mutants generation to obtain the faulty versions of the program which give the same high accuracy as that of the original program, hence difficult to identify the buggy behavior induced by them.
- Instead of relying on a synthetic data set (generated randomly) that may contain noise and missing values, we propose an MR to generate a new set of prioritized data without incurring additional labeling costs. Such MRs can be very useful in a scenario when we do not have enough test instances available but are still interested in verifying the system correctness over a diverse set of data (other than the data already available).
- Statistical analysis of probability scores over predicted classes (predicted by original and mutated programs) using a statistical hypothesis testing technique to check whether the difference is significant. If so, the buggy behavior is said to be successfully detected.
- Proposed an approach that takes advantage of ML techniques to test and predict faults in NN-based classifiers using prioritized test inputs.
- Conducted one more independent experiment to show that addition of an informative metadata features can further enhance the performance of proposed ML classifiers.

The rest of the paper is organized as follows. Section II talks about the related work. Section III discusses the motivation to use probability vectors/scores. Section IV presents the proposed approach and the contribution made in this paper. Section V discusses the experimentation and evaluation performed. Section VI discusses the threats to validity and, lastly, in Section VII, conclusions are drawn along with the future work.

II. Related Work

Testing ML applications is getting considerable attention in the research community, however, couple of challenges make it a harder task, i.e., (i) high labelling costs, and (ii) susceptibility to the oracle problem due to large input space. Byun et al. [16] focused on proposing a test prioritization technique to reduce the cost associated with labeling the data instances. They took advantage of the probability vectors (produced by either *softmax* or *sigmoid* output layers in neural networks) and proposed three sentiment measures (confidence,uncertainty, and surprise) to prioritize the inputs that are more likely to reveal faults in a trained model. Zhang et al. [17] used probability vectors to prioritize the uncertain or sensitive inputs that have the high potential to become adversarial examples using small perturbations. Dwarakanath et al. [6] proposed MT based approach to alleviate the oracle problem in testing DNN based image classifiers but the proposed approach is only applicable in an environment where the random seeds can be fixed to get deterministic results.

Shaikh et al. [1] introduced ML-based classification models (LibSVM and LinearLib) that use NASA dataset models to foresee the faults in a software defect-prone model. Based on comparative analysis performed, the results show that overall LibSVM attains high accuracy and is more efficient than LinearLib. Prabha et al., [2] first performed dataset dimensionality reduction (using PCA) and then applied Random Forest, Naïve Bayes, SVM, and Neural Networks to predict software defects that can help software developers to identify and correct a program's buggy code before deploying them in the actual environment. It also allows the software development team to prioritize and focus more on the modules that are problematic. Sethi [5] proposed Artificial Neural Network (a feed-forward

back propagation model) to predict defects in twenty software projects. It was found that the proposed model provides better performance than the classic fuzzy-based approach. Nehi et al., [8] took advantage of the history available in version control systems to perform software defect prediction. The authors used the code history information (e.g., bug reported by the client, code defects determined, etc.) extracted from several open-source projects and prepared a benchmark data set. To show its usefulness, the data set is used to evaluate the performance of different defect prediction approaches. The authors applied Artificial Neural Networks, Random Forest, K-Nearest Neighbor, Naïve Bayes, and Support Vector Machine and performed their evaluation using different measures e.g., Precision, Recall, F-measure, G-mean, and AUC. The results show that Random Forest outperforms and has higher predictive power in the identification of faults in the programs under test.

It is important to mention that the available literature primarily focuses on using the public data sets and harnessing ML approaches in the prediction of faults in traditional software, not specifically in ML applications. This makes it a potential area for exploration and making for fruitful contributions; thus we propose an approach that takes advantage of statistical and ML techniques to detect and predict buggy behavior in ML classifiers using prioritized test inputs.

III. Motivation To Use Probability Vectors / Scores

In this study, we took an advantage of using the probability vectors produced by the output layer in NN-based classifiers under test. Therefore, we provide a brief motivation regarding why we think that probability vectors contain additional information about the computation performed in an NN-based application and can be used for detecting and predicting buggy behavior in the programs under test. The probability vectors have been used to prioritize test inputs that are more likely to uncover faulty behavior in deep neural networks so that efforts can focus only on labeling the prioritized inputs [16]. These vectors help in deriving sentiment-based metrics (confidence, uncertainty, and surprise) that capture extra information about the computation performed inside deep neural networks on the given data input(s). The higher priority is given to the data inputs that are more uncertain or surprising (i.e., the inputs for which the probability distribution is spread out) and likely to reveal the faulty behavior in the trained model. The probability vectors have also been used to detect high-noise sensitive inputs that have a high potential to become adversarial examples, such that if a small perturbation is added to them, they can fool the DNN [17]. Alternatively, low noise-sensitive inputs will require a significant perturbation that will easily be detected by defensive models and hence is not effective.

Knowing that probability vectors have been used in solving the test input prioritization problem, we are interested in exploring their usefulness in the detection of buggy behavior and the development of ML models for its prediction in NN-based classifiers.

IV. Proposed Approach

This section discusses the proposed approach used for detection (using statistical hypothesis testing) and prediction (using ML-based approach) of implementation bugs in two NN-based classifiers under test, as shown in Fig. 1.

In order to detect and predict defects in NN-based classifiers under test, we took advantage of multiple techniques namely, (i) random sampling method [13] for selection of prioritized test inputs (ii) *MR*: applying valid transformation to the source/original prioritized inputs to generate new (follow-up) prioritized inputs without incurring additional labeling costs. (iii) *Mutants generation*: injecting high accuracy producing mutants into the NN-based classifiers under test, so that for the prioritized test inputs (both source and follow-up), the behavior of non-buggy/original and mutated program versions can be recorded and analyzed, (iv) *Statistical hypothesis testing*: to check statistically, whether the difference in the results produced by multiple program versions is statistically significant. If so, the buggy behavior is said to be detected, and (v) *ML*: automatic detection of bugs by leveraging ML techniques to learn from this difference and to predict whether, for the same prioritized test inputs, the new release under test shows the buggy or non-buggy behavior.

The effectiveness of the proposed approach is shown by testing two NN-based Network Intrusion Detection Systems (N-IDSs). The details for each classifier are shown in Table I.

- **Application#1:** This application is a shallow NN-based N-IDS (having one hidden layer, an input layer, and an output layer) that deals with a multi-class classification problem in an Open Stack environment and predicts whether the request is normal, by an attacker, or by a victim.
- **Application#2:** This application is a deep neural network-based N-IDS (having three hidden layers, an input layer, and an output layer) that deals with a binary class classification problem and predicts whether the network request is malicious or benign.

TABLE I
Classifiers Under Test Architecture

Classifiers Under Test	No of Layers	Hidden Layer(s) Type	Output Layer
Application#1	3	Fully Connected + ReLU	Softmax
Application#2	5	Fully Connected + sigmoid	Sigmoid

A. Step 1: Mutants Generation

For the development of a machine learning model, a data set containing information about the buggy/mutated and non-buggy/original program versions is needed. So, the first step is to obtain the buggy code for the classifiers under test. As we do not have access to actual buggy versions of the

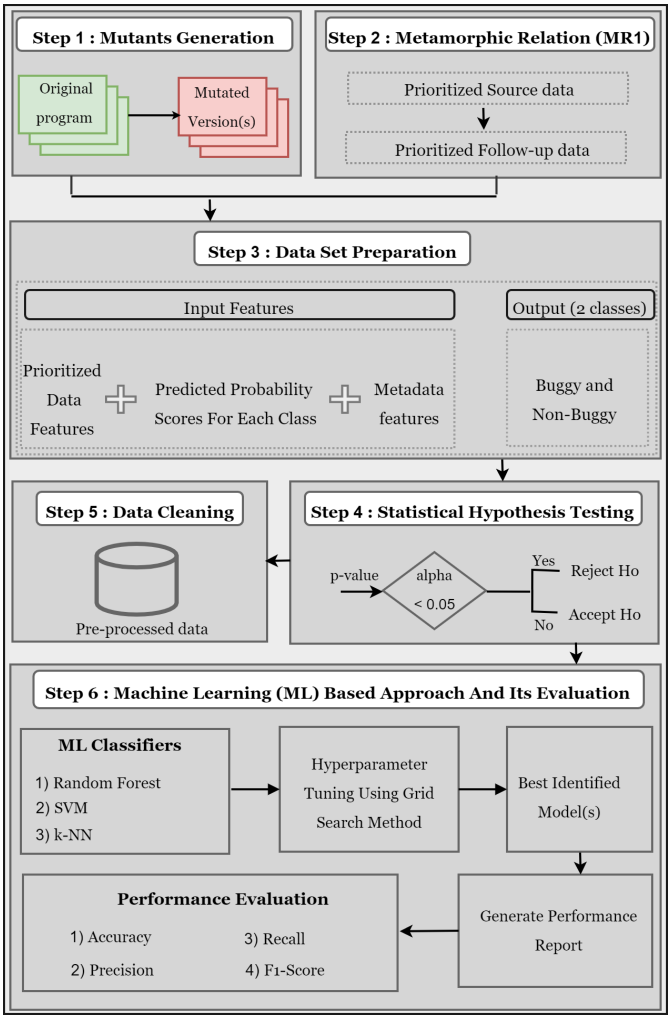


Fig. 1. Proposed Approach

classifiers under test, we took advantage of mutants generation process used in mutation testing technique [19], that is considered an effective approach in simulating the real faults made by programmers [18]. We injected 4 valid mutants into the programs under test that give the same high accuracy as that of the original program. These mutants belong to the category of CRP-constant replacement type mutants. The behavior recorded for all these mutated versions is treated as buggy behavior. As shown in Table II, it can be seen that these mutated versions achieve the same high accuracy as that of the original program, knowing the fact that they represent the faulty implementations. The goal is to apply statistical and ML techniques for the identification of buggy behavior induced by such high accuracy producing faulty programs; which otherwise may not be possible for software engineers who primarily focus on using accuracy measure to develop accurate models. Fig. 2 represents one of the injected mutants to simulate the mistake made by a software developer. It will cause the model to learn from a lesser number of informative features but it has been observed that this mutated model has produced almost the same high accuracy as that of the original program. The details of other mutants, the python code for the NN-based classifiers under test, the data used for training the models, the R-code for performing statistical analysis, and the proposed ML models' Python code, are all open-source and available online[1].

TABLE II
ORIGINAL AND MUTATED VERSIONS AVERAGE ACCURACY (%)

Application	Original	Mutant#1	Mutant#2	Mutant#3	Mutant#4
App#1	98.9%	98.3%	98.9%	N/A	N/A
App#2	92.5%	N/A	N/A	92.5%	92.9%

```
featuresToLocation = 42
X = traindata.iloc[:, 1:featuresToLocation]

To

featuresToLocation = 22
X = traindata.iloc[:, 1:featuresToLocation]
```

Fig. 2. Mutant#4: Original Code (top) and Mutant (below).

B. Step 2: Metamorphic Relation (MR-1) For New Test Inputs Generation- Shifting the features by constant k

Let X_{train} denote the training data and X_{test} denote the prioritized test data. In order to select the prioritized test instances, we have used the random sampling method [13]. After training the NN classifier, suppose it classifies a specific test instance x^i_{test} as belonging to class *'a'* (known as source execution). MR-1 says that if the features in both the training and test data are shifted by constant *k; where k = 0.3*, the output for the test instance x^i_{test} should remain the same i.e., class *'a'* (known as the follow-up execution). Such transformation will not change the existing relationship between the data points, and a correct NN model should learn from the relationship among the data points, not from their position in the space [6]. The follow-up transformation will help in the generation of new *'m'* prioritized follow-up instances from the given *'n'* number of prioritized original/source data instances, thus extending the prioritized test set size from *'n'* to *'n+m'*.

The proposed MR1 will facilitate in enhancing the testing phase effectiveness by doubling the size of prioritized test inputs, so that the behavior of NN-classifiers under test can be observed over a diverse set of data (both source and follow-up data). Another advantage of the proposed MR is that it also helps in removing the labeling costs for the newly generated test instances (follow-up data instances) that might be expensive in some cases.

C. Step 3: Data set Preparation

Due to the stochastic nature of NN-based classifiers, a single run may not be enough to verify their correctness, therefore, we obtain the results over multiple iterations. In this step, we record the results (probability distribution over classes

[1]https://github.com/matifkhattak/MLTesting/tree/master

predicted by original and mutated versions) and metadata information of NN-based classifiers under test separately for both the prioritized source and follow-up test instances. As shown in Fig. 1, each instance in the prepared data set is comprised of a prioritized test instance features, the probability scores over predicted classes, the trained NN Model's metadata information, and the class label (Buggy, Non-Buggy). As mentioned earlier, a correct NN classifier (retrained on the same data) may produce slightly different results for the given test instances; which does not necessarily mean that there is a bug in the system. Those multiple trained NN classifiers (if correct) may have different initialization points but will have almost the same convergence points [6]. By utilizing this property of NN classifiers, we obtain the results for prioritized test inputs over multiple iterations, whereas each iteration denotes a trained NN classifier. This allows us to capture the probability distribution for each prioritized test instance for a sufficient number of times that have been predicted by both the mutated and original NN-based classifiers under test.

D. Step 4: Statistical Hypothesis Testing

Once the probability scores (over predicted classes) for both the original/non-buggy and mutated/buggy programs are obtained, we compare the probability scores for each of the $class_i$ (produced by the mutated version) with the probability scores of the same $class_j$ (produced by the original version). This is important because before developing the ML model, it first needs to be confirmed that for the given prioritized test inputs, the difference between the probability distributions over predicted classes for both types of programs is statistically significant, otherwise, there is no information available for the ML model to distinguish the buggy program version's behavior from the non-buggy one. It is important to mention that the proposed approach is not meant to identify the individual injected mutants, instead we are interested to identify whether in general, the program behavior can be characterized as buggy or not. This is the reason that we treat the behavior produced by all the mutated versions as 'buggy' and formulate this problem as statistically comparing the two samples (buggy vs non-buggy). The following are the *Null and Alternative hypothesis*.

- H_0*:* There is no difference in the probability scores predicted by the original and the mutated programs.
- H_a*:* The probability scores predicted by the mutated programs are different from the original program.

After applying the test-statistic, the p-value obtained is compared with the set standard significance level (α = *0.05*). If it is less than the value of α, H_0 is rejected. The rejection of H_0 would suggest that (i) the difference between two samples is significant, showing that there is likely a bug in the system, and (ii) the probability scores do have some information available to distinguish both types of programs (buggy and non-buggy).

E. Step 5: Data Cleaning

It is possible that for multiple iterations the same probability distribution over classes is predicted for a specific test instance, which will result in the addition of duplicate observations in the data set. Such duplicate instances may bias the results of the ML models and can lead to incorrect conclusions. Therefore, a data cleaning process is an essential step in making sure that the observations are unique and the results produced by the models are unbiased. Thus, we performed the data preprocessing step and removed the duplicate observations before training the models.

F. Step 6: Proposed Machine Learning Based Approach

The last step is to harness ML techniques for predicting buggy behavior in NN-based classifiers using prioritized test inputs. For this purpose, we select three widely used ML algorithms in the literature (i.e., Random Forest, SVM, and k-NN) to extract the hidden knowledge captured in the data set, as shown in Fig. 1. Due to the space limit, we do not discuss them here but we refer the interested readers to read [14] for details. The data set is split into 80%-20%, where 80% is used for training purposes and 20% is used for testing. We use the k-fold cross-validated grid-search method to find the best hyperparameter settings for the classifiers. The best classifier (with optimal hyperparameter settings) is identified and is used as a final model to predict the output for the test data. In the end, a performance report is generated for each of the classifiers, and results are compared using the evaluation metrics e.g., accuracy, precision, recall, and F1-score (harmonic mean of precision and recall) [14], to select the best model having higher prediction power.

V. Experimentation and Evaluation

By taking into account our proposed approach, we derived the following three Research Questions (RQs):

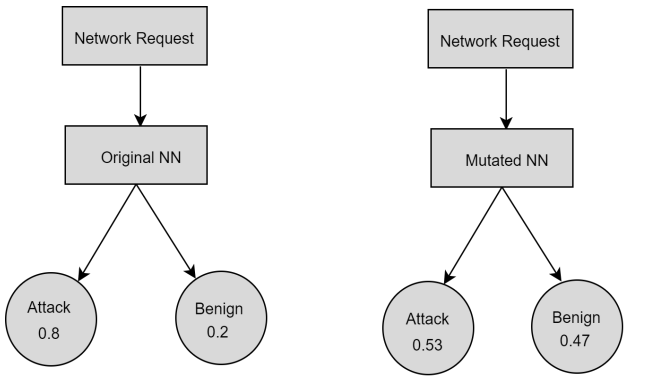
- **RQ1**: Is the proposed statistical hypothesis testing technique effective in detection of buggy behavior in the classifiers under test?
- **RQ2**: Is the proposed ML-based approach effective and which ML model is more suitable for the problem under investigation?
- **RQ3**: Does the addition of metadata features increase the performance of proposed models?

A faulty new program release giving the same high accuracy as that of the original/correct program may classify a test instance with the same class label 'a' that was predicted by the original/correct program but may affect the probability distribution over the predicted classes. As an example, Fig. 3 shows that both the original and high accuracy producing mutated programs have predicted the same class label 'attack' but the probability distributions over predicted classes have significantly changed for the mutated version. Our experiment shows that unlike predicted class labels, the probability scores provide more granular details and can be analyzed statistically to detect the buggy behavior induced by such high accuracy producing mutated program versions. In total, we injected 4 valid mutants into the programs under test and performed multiple iterations to obtain the results for 200 prioritized test instances (100 source and 100 follow-up instances), which

resulted in a total of 8000 data instances (50% of the data is labeled as buggy and the other 50% is labeled as non-buggy). The data set for App#1 contains 37 features, whereas, for App#2, it contains 46 features.

RQ1: Is the proposed statistical hypothesis testing technique effective in detection of buggy behavior (produced by the high accuracy producing mutated program versions) in the classifiers under test?

Before training the ML model, it is important to check whether the probability distribution over predicted classes for both types of programs (original and mutated) is statistically significant. Therefore, we perform statistical hypothesis testing to check whether, for the given prioritized test inputs, the injected mutants have significantly changed the probability distributions over predicted classes. Comparing the distributions of two classes (e.g., $class_i$ probability scores produced by mutated code vs the same $class_j$ probability scores produced by the original code) can be treated as a problem of statistically comparing two samples. During a single run, the same model is used for predictions for multiple prioritized test inputs, so they all are connected to the same model. For this reason, applying a paired t-Test will be an appropriate choice. However, during analysis, we found that the normality assumption was badly violated, hence applying the paired t-Test, although known to be a robust test, may not provide reliable results. For example, Fig. 4 provides evidence that the normality assumption is violated for one of the classes (because of large tails at both ends). For this reason, the *Wilcoxon signed-rank test* is applied, which is a non-parametric test used for paired data and does not rely on the satisfiability of the normality assumption [13].



(a) Original program (b) High accuracy producing mutated version

Fig. 3. Final predicted output (i.e., attack) is same but probability distributions over predicted classes have significantly changed for the mutated program

After performing the statistical analysis on the probability distributions over predicted classes, results in Table III and Table IV show that we have strong evidence to reject H_0 for all the classes (except for class2 in App#1) for both the classifiers under test (p-value < 0.05). Rejection of H_0 means that the difference is significant and there is likely some bug in the system, hence the buggy behavior is said to

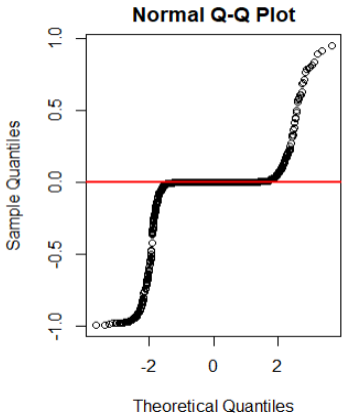


Fig. 4. Q-Q plot

be detected. It is important to note that the mutated program versions under investigation produce the same high accuracy as the original program, knowing the fact that they represent the faulty implementation of the programs under test. Such mutants may otherwise be difficult to detect and can mislead if the software developers solely rely on observing the accuracy of the ML model in a new release. **Therefore, we conclude that high accuracy does not necessarily mean that the program is correct/free of bugs, and the proposed statistical hypothesis testing technique is effective in the detection of buggy behavior produced by such high accuracy producing program versions, which answers RQ1**.

TABLE III
WILCOXON SIGNED RANK TEST RESULTS FOR APP#1 ($\alpha = 0.05$)

Class Label	p-value	Reject H_0	Buggy behavior Detected?
Class1 (Normal)	< 0.0001	Yes	Yes
Class2 (Attacker)	0.63	No	No
Class3 (Victim)	< 0.0001	Yes	Yes

TABLE IV
WILCOXON SIGNED RANK TEST RESULTS FOR APP#2 ($\alpha = 0.05$)

Class Label	p-value	Reject H_0	Buggy behavior Detected?
Class1 (Attack)	< 0.0001	Yes	Yes
Class2 (Benign)	< 0.0001	Yes	Yes

RQ2: Is the proposed ML-based approach effective and which ML model is more suitable for the problem under investigation?

After answering RQ#1 and finding that the difference between the probability distribution over predicted classes is statistically significant, the next step is to use this knowledge for developing the ML models, so that the 'buggy' and 'non-buggy' behavior for the prioritized test inputs can be learned and then using this knowledge to predict the faulty behavior in the new release (using the same prioritized test inputs). Table V and Table VI shows the performance of Random Forest,

SVM (with RBF kernel) and k-NN algorithms on App#1 and App#2 data sets respectively. We used a fixed random seed to make sure that the data set is split in the same way for all the algorithms, so that, every algorithm is trained and evaluated on the same data instances. The evaluation metrics used to evaluate the performance of models include accuracy, precision, recall, and the F1-score. The results obtained show that for both the classifiers under test, random forest outperforms, having higher scores for all four measures. For App#1, SVM and k-NN provide satisfactory performance but not very well for App#2. Apart from that, SVM attains a lower F1-score than k-NN for App#2 but performs better than k-NN for App#1. We also observed that for the given data set, even the best identified k-NN model (using 10-fold cross-validated grid-search method) is over-fitting, thus may not be an appropriate choice for predicting buggy behavior in App#2. Based on the results shown in Table V and Table VI, we conclude that overall, the proposed ML models have performed well in extracting the hidden patterns of 'buggy' and 'non-buggy' program versions. Furthermore, among the proposed models, random forest outperforms and attains higher performance (high accuracy and F1-score) than SVM and k-NN. **Hence, it answers our RQ2 that the proposed ML based approach is effective in predicting the faulty behavior in the programs under test (especially for App#1), and among the proposed models, Random Forest achieves the best performance for both the NN based N-IDSs under test**.

TABLE V
PERFORMANCE REPORT ON APP#1 DATA SET

Classifier	Accuracy	Precision	Recall	F1
SVM	0.87	0.86	0.88	0.87
Random Forest	**0.93 ± 0.0**	**0.93 ± 0.0**	**0.93 ± 0.001**	**0.93 ± 0.0**
k-NN	0.84	0.83	0.86	0.85

TABLE VI
PERFORMANCE REPORT ON APP#2 DATA SET

Classifier	Accuracy	Precision	Recall	F1
SVM	0.60	0.67	0.47	0.55
Random Forest	**0.83 ± 0.004**	**0.86 ± 0.008**	**0.81 ± 0.008**	**0.83 ± 0.005**
k-NN	0.59	0.62	0.54	0.58

RQ3: Does the addition of metadata features increase the performance of proposed ML models?

The results in Table VI show that for App#2, the SVM, and k-NN classifiers do not achieve very good performance. This motivates us to conduct one more experiment and to add metadata features extracted during training of both types of programs (original and mutated ones), expecting the addition of new informative features to increase the performance of models. The two metadata features extracted include *'training time taken in minutes'*, and *'training time taken in seconds'*. We are interested in observing whether the addition of metadata features can further discriminate both classes (high accuracy producing buggy code and non-buggy code) and whether they can help to enhance the ML models' performance. This model can be used in a scenario where the developer has submitted the change documentation mentioning that the change is neither related to enhancing/reducing the model complexity nor significantly expanding the training set size. An example can be: changing the Min-Max Normalization method to z-Score Normalization. Such change should not have a significant impact on either enhancing or reducing the model's training time. However, if the model mistakenly becomes overly or insufficiently complex, it will have an impact on the model training time which can be a useful feature in capturing the information about the trained NN-based classifier under test. Based on the results obtained on a new data set having metadata features, if the results in Table VII are compared with Table VI, it can be seen that the performance of all the models has significantly improved. In comparison to k-NN, SVM attains high accuracy and precision but has a low recall value. However, random forest still outperforms and achieves the best predictive power. **Therefore, we conclude that the addition of metadata features is useful in enhancing the ML models' performance and can play a handy role in developing accurate ML models, which answers RQ#3**.

TABLE VII
PERFORMANCE REPORT ON APP#2 DATA SET AFTER ADDING METADATA FEATURES

Classifier	Accuracy	Precision	Recall	F1
SVM	0.76	0.96	0.54	0.69
Random Forest	**0.90 ± 0.0**	**0.96 ± 0.0**	**0.93 ± 0.01**	**0.94 ± 0.01**
k-NN	0.70	0.72	0.65	0.68

VI. THREATS TO VALIDITY

In this study, we do not aim to identify the individual injected mutants, instead we are interested in identifying whether, in general, the program behavior can be characterized as buggy or not. This is the reason that we treat the behavior produced by all the mutated versions as 'buggy' and by the original program as 'non-buggy'. The proposed approach serves its purpose well in meeting this objective. Nevertheless, we identify the following threats to validity:

- Although we propose only a single MR, it worked sufficiently well to show the applicability of the MT approach, (i) in extending the prioritized test set with additional tests without incurring additional labeling costs, and (ii) using the prioritized test inputs (both source and follow-up data) to record and observe the behavior of the classifiers under test over a diverse set of inputs.
- Due to time and resource constraints, we are only able to find and inject 4 valid high accuracy producing mutants in the NN-based N-IDSs under test. Although the number of mutants is small, it fulfills the objective and applicability of the proposed approach in detecting their buggy behavior that may otherwise be difficult to detect by a software developer. Second, the main reason to use only the high accuracy producing mutants is that

when the software developer sees the low accuracy of the model, the developer gets an alert indicating that some problem in the model requires further investigation. However, when the model produces high accuracy, that may be skeptical and mislead the developer, assuming that everything is fine and model is ready to be deployed in the production environment.

- It can be argued that some of the mutants will produce a different probability class distribution than the original/correct program version, and hence it can be easy to identify them. This poses a potential threat to the construct validity of this study, however, this observation is subjective and is error prone. Also, it may not hold true without providing some solid statistical evidence. Second, in order to automate the testing process, there must exist a systematic approach to automatically detect the buggy behavior in such high accuracy producing faulty program versions.
- The NN-based N-IDSs under test belong to the class of fully connected neural networks, thus generalizing the results to other types of DNNs e.g. CNNs, RNNs, would be speculative and reveals a threat to external validity.

VII. Conclusion And Future Work

The high labeling cost associated with data instances and testing computationally complex machine learning classifiers that have stochastic behavior are both challenging and resource-intensive tasks. To address the first challenge, we propose a Metamorphic Relation (MR) that effectively solves the data generation/labeling problem without any need to label the new test instances manually. To target the second challenge, we propose a statistical hypothesis testing (for detection) and machine learning-based approach (for prediction) of faulty behavior in NN-based classifiers using the prioritized test inputs. The proposed statistical and ML-based approach is applicable for testing NN-based classifiers in an environment where the random seeds can not be fixed for getting deterministic results and checking the program correctness. The usefulness of our proposed approach is shown in detecting and predicting buggy behavior in two NN-based network intrusion detection systems i.e., one based on Shallow NN, whereas the other is a DNN-based classifier. The results obtained show that, (i) the proposed statistical hypothesis testing is effective in detecting the buggy behavior, and (ii) among the proposed ML models, random forest outperforms and achieves better performance than SVM and k-NN algorithms.

In this paper, we propose a sample MR to show the effectiveness of MT in solving the data generation/labeling problem. In future research work, we intend to propose more effective MRs that can be used to generate additional representative data; thus reducing the labeling cost further and enabling organizations to check program correctness on large input scenarios for enhancing their trust. It will also be interesting to explore what other types of useful features can be added to enhance the performance of ML models and to show the applicability of the proposed approach in general. The results obtained in this preliminary study are encouraging, and a comprehensive study on applying the proposed approach on a larger number of mutants is in progress.

References

[1] Shaikh, S., Changan, L., Malik, M. R., & Khan, M. A. (2019, December). Software Defect-Prone Classification using Machine Learning: A Virtual Classification Study between LibSVM & LibLinear. In *2019 13th International Conference on Mathematics, Actuarial Science, Computer Science and Statistics (MACS)* (pp. 1-6). IEEE.

[2] Prabha, C. L., & Shivakumar, N. (2020, June). Software Defect Prediction Using Machine Learning Techniques. In *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)* (pp. 728-733). IEEE.

[3] Weyuker, E. J. (1982). On testing non-testable programs. *The Computer Journal*, 25(4), 465-470.

[4] Chen, T. Y., Cheung, S. C., & Yiu, S. M. (2020). Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543*.

[5] Sethi, T. (2016, September). Improved approach for software defect prediction using artificial neural networks. In *2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)* (pp. 480-485). IEEE.

[6] Dwarakanath, A., Ahuja, M., Sikand, S., Rao, R. M., Bose, R. J. C., Dubash, N., & Podder, S. (2018, July). Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 118-128).

[7] Li, Z., Cui, Z., Liu, J., Zheng, L., & Liu, X. (2020, January). Testing Neural Network Classifiers Based on Metamorphic Relations. In *2019 6th International Conference on Dependable Systems and Their Applications (DSA)* (pp. 389-394). IEEE.

[8] Nehi, M. M., Fakhrpoor, Z., & Moosavi, M. R. (2018, May). Defects in The Next Release; Software Defect Prediction Based on Source Code Versions. In *Electrical Engineering (ICEE), Iranian Conference on* (pp. 1589-1594). IEEE.

[9] Rehman, F., & Izurieta, C. (2021, July). Statistical Metamorphic Testing of Neural Network Based Intrusion Detection Systems. In *IEEE International Conference on Cybersecurity and Resilience (CSR)* (In press).

[10] Ohnsman, A. (2018). Lidar maker velodyne 'baffled'by self-driving uber's failure to avoid pedestrian. *Forbes, March*.

[11] Lambert, F. (2016). Understanding the fatal tesla accident on autopilot and the nhtsa probe. *Electrek, July, 1*.

[12] Ziegler, C. (2016). A google self-driving car caused a crash for the first time. *The Verge.*

[13] Ramsey, F., & Schafer, D. (2012). *The statistical sleuth: a course in methods of data analysis*. Cengage Learning.

[14] Alpaydin, E. (2020). *Introduction to machine learning*. MIT press.

[15] Sun, L., & Zhou, Z. Q. (2018, November). Metamorphic testing for machine translations: MT4MT. In *2018 25th Australasian Software Engineering Conference (ASWEC)* (pp. 96-100). IEEE.

[16] Byun, T., Sharma, V., Vijayakumar, A., Rayadurgam, S., & Cofer, D. (2019, April). Input prioritization for testing neural networks. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)* (pp. 63-70). IEEE.

[17] Zhang, L., Sun, X., Li, Y., & Zhang, Z. (2019). A noise-sensitivity-analysis-based test prioritization technique for deep neural networks. *arXiv preprint arXiv:1901.00054*.

[18] Smith, B. H., & Williams, L. (2007, September). An empirical evaluation of the MuJava mutation operators. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)* (pp. 193-202). IEEE.

[19] Jia, Y., & Harman, M. (2010). An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5), 649-678.