Statistical Metamorphic Testing of Neural Network Based Intrusion Detection Systems

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

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

2021 IEEE International Conference on Cyber Security and Resilience (CSR) | 978-1-6654-0285-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/CSR51186.2021.9527993

***Abstract*—Testing computationally complex neural network-based applications (i.e. network intrusion detection systems) is a challenging task due to the absence of a test oracle. Metamorphic testing is a method to potentially solve the oracle problem when the correctness of individual output is difficult to determine. However, due to the stochastic nature of these applications, multiple runs with the same input can produce slightly different results; thus rendering traditional metamorphic testing technique inadequate. To address this problem, this paper proposes a statistical metamorphic testing technique to test neural network based Network Intrusion Detection Systems (N-IDSs) in a non-deterministic environment. We also performed mutation analysis to show the effectiveness of the proposed approach. The results show that the proposed method has a strong defect detection capability and is able to kill 100% implementation bugs in two neural network-based N-IDSs, and 66.66% in a neural network-based cancer prediction system.**

***Keywords*—Metamorphic testing, oracle problem, statistical hypothesis testing, intrusion detection system, neural networks, stochastic algorithms**

I. Introduction

Information Technology (IT) practitioners grapple on a daily basis with how to maintain their networks secure from malicious adversaries. A large number of tools exist today that can help with identifying potential weaknesses and vulnerabilities regarding all types of Cybersecurity concerns. The ISO 25k standard [6] characteristics helps us partition such threats into different categories. However, operational solutions to theoretical characterizations are not contextual, and require significant manual efforts from practitioners to identify relevant attacks. To aid practitioners and prevent such disastrous threats proactively, one possible solution would be to choose and deploy an intelligent machine learning based Network Intrusion Detection System (N-IDS). These automated techniques act in context and remove significant manual efforts. A challenge faced by these systems, however; is how can we trust and rely on the correctness of such computationally complex machine learning based N-IDSs?, especially, when the organization has purchased it from a new vendor or built on top of some open source libraries.

Machine learning (ML) is heavily used in solving real-world problems in many application domains like finance, healthcare, transportation, machine translation, voice recognition, safety-critical applications (e.g., self-driving cars and self-flying drones), and as exemplified above, the Cybersecurity domain. Normally, we focus more on the development of accurate models but much less on ensuring their quality. Thung et al. [4] showed in their study that 22.6% of faults in ML applications are due to incorrect implementation that caused them to produce inconsistent and unexpected results. A small bug in the system may lead to catastrophic failure which can result in both financial and human loss. For example, on 14 Feb 2016, a Google self-driving car crashed due to misjudgment and putting itself into the path of an oncoming bus, in an attempt to avoid sanbags [1]. In May 2016, a Tesla Model S crashed when the autonomous driving system hit the trailer and did not treat it as an obstacle [2]. In March 2018, an Uber self-driving car killed a woman in Arizona, when at night it failed to recognize a pedestrian on the road [3]. In the face of these disasters, ensuring the correctness of ML-based systems is very challenging but equally an essential problem to be solved.

It is important to note that in comparison to traditional software systems, testing ML applications poses several challenges. First, when we talk about traditional software solutions, they are manually programmed to perform the needed functionality. The code of program under test is usually fixed and the output produced by the program is generated using the specified set of rules. However, the rules defined in ML applications are not explicitly hard-coded, instead, they involve complex computations and are learned from the training data. Second, ML applications have a large input space for which they need to be verified. Features like price, date of birth, width, height, and road conditions (in an image) may contain a large range of values. As a result, it is difficult to verify the correctness of a system against all those possible values. Third, the low accuracy of ML models is a composite effect, which can arise from a combination of three ML components, namely: data, program (code written by the programmer), and the framework/library (e.g., Weka, Pytorch, TensorFlow), whereas each of them many contain bugs. Therefore, it is essential to verify the correctness of each of these ML components. Finally, testing ML-based applications seriously suffer from the *Oracle problem* due to the difficulty in assessing

whether the generated output is correct [5]. An oracle is a mechanism where a program is verified by comparing its output with the expected outcome. To test such complex systems, either the oracle is unavailable or too expensive to apply.

A major approach used to solve the oracle problem in such non-testable programs is known as Metamorphic Testing (MT) [7]. MT has been proven to be an effective approach in alleviating the oracle problem in testing ML applications, for which the correctness of individual output is difficult to determine [8] [9]. The first step in metamorphic testing is to identify the set of necessary properties/characteristics of a system known as Metamorphic Relations *(MRs)*. Each MR describes the relationship between the input and the related output that specifies how the output of the program should be changed when changing the input. A simple example of an MR for a program calculating the standard deviation of a list of numbers can be stated as *'shuffling the order of elements in a list should not change the final output'*. Once the MRs have been identified, *source* and *follow-up* test cases are generated. The *source test case* (treated as original data) can either be domain-specific or can be randomly generated, whereas the *follow-up* test case is generated by performing some valid transformation to the original data (as specified in the related MR). Both the *source* and *follow-up* test cases are then executed on the target program under test. If the results generated by the source and follow-up executions do not adhere to the relation specified in the MR, then this would indicate a potential bug in the system.

One of the challenges faced in applying MT to Neural Network (NN) based applications is their stochastic nature (due to random initialization of weights) where they produce slightly different outputs for multiple runs with the same inputs. In order to get consistent results for both the source and follow-up test cases, a couple of researchers have proposed the solution of fixing the random seeds [10][11]. The first paper is focused on applying MT to uncover implementation bugs in a Deep Learning (DL) based image classifier [10]. The authors used maximum standard deviation σ_{max} (based on variation in a loss on the test data) as a threshold to verify if the program under test adhered to MRs or not. In order to alleviate stochasticity, the authors fixed the random seed to get deterministic results (obtaining consistent σ_{max}) for the program under test. The authors highlighted that they were able to get deterministic results on CPUs but not on GPUs due to inherent non-determinism introduced by NVidia CUDA libraries. Hence, their approach was limited to work only on CPUs. The second paper deals with applying MT to test an Artificial Neural Network (ANN) based classifier, taken from Stanford's cs231n course [11]. In order to get deterministic results and to verify the MRs using the equality operator, the authors initialized the weights of a classifier with fixed values. The problem with this approach is two-fold, (i) it will not work in an environment where either the weights of the ANN cannot be fixed or where getting non-deterministic results is unavoidable, and (ii) it may not be applicable in a scenario when a model needs to be trained on GPUs to accelerate the training time.

In order to solve the above highlighted problems, this paper proposes a statistical-based MT technique to unveil implementation bugs in NN-based N-IDSs, especially, in an environment where neither the random seeds nor the weights can be fixed in order to get deterministic results. Apart from that, in real-world it may not be possible for a software tester to fix the random seeds explicitly in a project that has millions lines of code and is also using a large number of third-party libraries. Therefore, instead of relying on a single run, the proposed approach statically analyses the results over multiple iterations (each iteration denotes a trained NN classifier) because a correct NN classifier should converge to the same solution most of the time, if not always [10]. To show the applicability of the proposed approach, we have worked with three ML applications. Two applications belong to Cybersecurity space i.e. N-IDSs, whereas the third one is from the healthcare domain that classifies cancer types among patients.

The following are the main contributions of this paper:

- Three metamorphic relations are proposed to uncover implementation bugs in ML-based applications (i.e., N-IDSs and Cancer prediction system).
- Four statistical measures are used that will allow software testers to verify the correctness of a program under test (especially in a non-deterministic environment) using a combination of statistical hypothesis testing and MT technique.
- Mutation testing [12] is applied to show the effectiveness of the proposed MT-based approach. The results show that the proposed method is able to kill 100% implementation bugs in the two N-IDSs, whereas 66.66% in the cancer prediction system.

The rest of the paper is organized as follows: First, the related work is presented in Section II. Next, the proposed approach for testing N-IDSs is presented in Section III. Section IV presents the results, showing the effectiveness of our proposed approach, and Section V concludes the paper along with potential future work.

II. Related Work

Deploying applications that are not fully tested can have disastrous consequences in the real world. Zhou et al. [9] reported serious defects in the Uber system 8 days prior to when the autonomous Uber killed Elaine Herzberg (a pedestrian) on March 18, 2018, in Tempe, AZ. The authors applied Fuzz testing in combination with MT to test LiDAR (light detection and ranging), an obstacle perception module used in Uber, and revealed several previously unknown fatal errors. Xie et al. [8] applied MT (as a test oracle) to test a popular open-source ML tool, known as Weka. Weka provides a large number of algorithms for data-prepossessing, classification, clustering, prediction, feature selection, and visualization [13]. The proposed MRs are not only able to find implementation bugs in K-Nearest Neighbors and Naive

Bayes classification algorithms (treated as a verification step) but are also helpful in serving as validation steps. Pei et al. [14] proposed DeepXplore, a white-box testing framework to address two main challenges faced in the automated testing of a large-scale DL system: (i) identification of erroneous behavior(s) in the system without a need to labeling each test instance manually, and (ii) generation of inputs that can exercise different parts of a DL system's logic (known as neuron coverage) to uncover hidden defects in it. The proposed approach has shown promising results in uncovering thousands of erroneous behaviors in 5 DNNs. However, one of the limitations of the proposed approach is its strong dependency on a differential testing technique, which requires at least two similar implementations of a system. Apart from that, the authors did not provide any details regarding how they verified that the generated images truly represent real-world scenarios.

Normally, practitioners rely on using accuracy measures to check the appropriateness of an algorithm for the underlying problem. Li et al. [11] showed that higher accuracy does not necessarily mean that the trained model is free of bugs. The authors applied the MT technique to test NN classifiers. The proposed MRs first transform the original inputs to generate follow-up test cases and then check, whether the produced output (for both source execution and follow-up execution) adheres to the corresponding MRs or not. To check the effectiveness of the proposed MRs, mutation testing is applied, and artificial bugs are injected into the source code of a program under test. The authors highlighted a few mutants that produced the same high accuracy as that of the original program, knowing the fact that they represent the faulty implementations. The results obtained show that the proposed MT-based approach is more effective in detecting faults than using the accuracy measure. However, the proposed approach is only applicable in an environment where the software tester can fix the random seeds to get deterministic results, which is not always possible.

It is important to note that despite efforts to find relevant references in the literature that address how MT could be used to test N-IDSs, we were unsuccessful, and although beyond the scope of this paper, we believe this is an important gap in the body of knowledge that needs further investigation.

III. Proposed Approach

In this section, we present the proposed statistical-based MT technique in detail. In order to verify the proposed approach, two research questions are raised in this paper.

- **RQ1**: How effective is statistical MT in identifying the faults in NN-based classifiers when trained in a non-deterministic environment?
- **RQ2**: Do all MRs have the same defect detection ability for NN-based applications?

First, we provide three MRs which are used to find implementation bugs in the applications under test. Next, instead of checking the correctness of MRs over a single source and follow-up execution (not possible due to random initialization of weights causing the non-deterministic behavior), we obtain results over multiple iterations and analyze them statistically (using the proposed statistical measures) to verify whether the outputs adhere to the relation specified in the MRs or not. The violation of an MR will be an indication of a potential bug in the program under test. Lastly, a mutation testing technique is applied on the following N-IDSs to check the effectiveness of proposed MRs. In order to show the relevance of the approach in other domains (i.e., health care), we also include a DNN-based cancer prediction system. The details for each application are presented in Table I.

- **Application#1:** A shallow NN-based N-IDS used for the detection of malicious attacks in an OpenStack environment. It is a multi-class classification problem that classifies the network request among 3 class labels (*normal*, *attacker*, and *victim*).
- **Application#2:** A DNN-based N-IDS for intrusion detection in a network that solves a binary class classification problem (i.e., either the request is *attack* or *benign*).
- **Application#3:** A DNN-based cancer prediction system used to identify cancer types among patients. It is a multi-class classification problem that classifies the patient among one of 10 cancer types.

TABLE I
Neural Network Model Architectures

Application	# Hidden Layers	Hidden Layer(s) Type	Output Layer
Application#1 (ANN N-IDS)	1	Fully Connected + ReLU	Softmax
Application#2 (DNN N-IDS)	3	Fully Connected + sigmoid	Sigmoid
Application#3 (DNN Cancer identification)	3	Fully Connected + sigmoid	Softmax

A. Metamorphic Relations (MRs)

We propose the following three MRs that are applicable to all three ML based applications under test:

1) MR-1:- Changing the order of features (of both training and test data): Let X_{train} be the training data and X_{test} be the test data. After training the neural network, let a specific test instance x^i_{test} be classified as class *'a'*. MR-1 says that if we change the order of attributes in both the training and test data, the output for the test instance x^i_{test} should remain same (i.e., class *'a'*).

2) MR-2:- Addition of uninformative attribute to both training and test data: Let X_{train} be the training data and X_{test} be the test data. After training the neural network, let a specific test instance x^i_{test} be classified as class *'a'*. MR-2 says that if we add an uninformative attribute (attribute having value 0) to all the instances of both the training and test data, the output for the test instance x^i_{test} should remain same.

3) MR-3:- Shifting of both the training and test features: Let X_{train} be the training data and X_{test} be the test data. After training the neural network, let a specific test instance x^i_{test} be classified as class *'a'*. MR-3 says that if we shift the

features in both the training and test data with some constant *c*, it will not change the existing relationship between the data points, so the output for the test instance x^i_{test} should remain same.

B. Statistical Hypothesis Tests

In this section, four statistical measures are discussed that we used to perform statistical hypothesis testing and verification of a relation specified in MRs. For this purpose, we added custom code in the classifiers under test to store their predicted results in excel files. The generated excel files contain detailed information for each test instance e.g. expected class label, predicted class label, the probability distribution for multiple classes, the maximum probability for the predicted class label, etc. We then developed a C# utility to process those excel files and extract the needed information on which the proposed statistical measures are applied. If the results are found statistically significant e.g., if the probability (*p-value*) is less than the set significance level (α), we reject the *Null Hypothesis (H_0)*. Rejection of H_0 would suggest that the MR has been violated and that there is a potential bug in the system. The classifiers' original code, mutants generated, excel files (containing predictions) and their processed versions, the C# utility, and the results produced, all are open-sourced [1]). The statistical measures are as follows:

1) Maximum Voting: To understand the proposed maximum voting concept, let X_{sData} be the source data, and X_{fData} be the follow-up data. We train the same classifier *'n'* times on the source data and *'m'* times on the follow-up data (where *n = m*). This results in C_n trained classifiers on the source data and C_m trained classifiers on the follow-up data. Those trained models (C_n and C_m) are then used to predict the class label for the test instance x^i_{test}. The results obtained are accumulated and a frequency distribution table is prepared, as shown in Table II.

For a given test instance x^i_{test}, the MR is said to be violated, if the maximum times of the class predicted for the source executions is different than the maximum times predicted for the follow-up executions.

2) Comparing Distributions Using Chi-square Test of Homogeneity & Fisher's Exact Test: There may exist some scenarios where the *Maximum Voting* approach may not work. For example, in Table II, it can be seen that for both the source and follow-up executions, the class which is predicted maximum times is *Class2*. So, the proposed measure would suggest that the MR is satisfied. However, one may argue that the difference between the *Class2* and *Class3* distribution (for follow-up executions) is not very high and that this difference could be treated as the identification of a potential bug in the system. This motivates us to analyze the distributions over multiple class labels for better identification of *true positive* alarms. For this purpose, we take advantage of the Chi-square test of homogeneity ($\tilde{\chi}^2$), which is used to compare two samples having unknown population distributions. Using the frequency table (as shown in Table II), we formulate the underlying problem as one where we compare the source executions' distribution S_i (treated as the expected distribution) with the follow-up executions' distribution F_i (treated as the observed distribution).

$$\tilde{\chi}^2 = \sum_{i=1}^{n} \frac{(F_i - S_i)^2}{S_i}$$

However, one of the limitations of the Chi-square test is that it may produce inaccurate and unreliable results if any of the cell values is less than 5 [15]. To solve this problem for some of the distributions we obtained, we apply an alternative test known as Fisher's exact test, which works equally well for the distributions having small cell values [15]. Based on the results obtained, if the p-value is less than the set significance level, *null hypothesis* is rejected, which ultimately means that the MR is violated and there is some potential bug in the system.

The following are the proposed *null* and *alternative hypothesis* used for both the $\tilde{\chi}^2$ and Fisher's exact test.

- H_0*:* The distributions for both the source and follow-up executions are same.
- H_a*:* The distribution for the source executions is different from the follow-up executions.

TABLE II
FREQUENCY DISTRIBUTION TABLE

Execution Type	Class1	Class2	Class3
Source (n=30)	3	15	2
Follow-up (m=30)	3	9	8

3) Comparing Distributions Using Two Sample t-Test & Permutation Test: In an NN-based classifier, an activation function (e.g., sigmoid or softmax) is used in the output layer that generates a probability vector, providing the probability for each of the class labels. A class with the highest probability is treated as the predicted class label for the given test instance. We take advantage of using these probabilities and perform statistical analysis to check how close the probability distributions are for both the source and follow-up executions for any given instance.

We treat this problem as comparing the two sample means using the t-Test. First, the results in excel files are processed (using the C# program) and the class label predicted maximum times during the source executions is identified. For the given test instance x^i_{test}, the purpose is to first find the class label for which the model is more confident and then extracting the probability of that specific class for both the source and follow-up executions. This will result in the generation of two samples for each MR, one for the source and other for the follow-up executions, as shown in Table III.

Let x = (x_1, x_2,...,x_n) and y = (y_1, y_2,...,y_n) represent two samples, one corresponding to source executions and the other corresponding to follow-up executions. In order to conduct the statistical t-Test, we need to find the mean $\bar{x_n} = \sum_{i=1}^{n} x_i$ and the variance $s^2_{x,n} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x_n})^2$ for both samples. The t-score is calculated as,

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

TABLE III
EXEMPLARY PROBABILITIES

Source Executions	Follow-up Executions
0.63	0.61
0.64	0.60
0.61	0.62
0.61	0.65
0.60	0.59
0.61	0.60

$$t = \frac{\bar{x_n} - \bar{y_n}}{\sqrt{\frac{s^2_{x,n}}{n} + \frac{s^2_{y,n}}{n}}}$$

After applying the t-Test, the p-value obtained is compared with the significance level. If it is less than the set significance level, H_0 will be rejected, which ultimately means that the MR is violated and that there is a potential bug in the system. The following are the *null* and *alternative hypothesis* to check whether the MRs have been satisfied or not.

- H_0*:* There is no difference in the sample means of both the source and follow-up executions.
- H_a*:* The sample mean of source executions is different from the sample mean of follow-up executions.

It is important to note that before applying the two-sample t-Test, we have analyzed the data to check whether the assumptions are fully satisfied. During the analysis (using a Q-Q plot), we found that for some of the MRs, the normality assumption is slightly violated.

However, the t-Test is robust against such violations and can still be applied [16]. To perform better analysis and support decision-making, we show our results using both the t-Test and the permutation test in §IV. The permutation test is a non-parametric test that does not rely on the normality assumption. For more details about the permutation test, we refer the interested readers to read Chapter 1 [16].

IV. EMPIRICAL RESULTS

All three applications (N-IDSs and Cancer identification system) under test are built on Keras 2.3.1 and TensorFlow 2.0. We used the MutPy [17] tool to generate mutated versions of the classifiers under test. After excluding the mutants, which were either changing the architecture of the neural network or causing the program to crash, we selected 3 valid mutants for each application (details are available online [2]). One of the generated mutants is shown in Fig. 1. A mutant is said to be killed if the results do not adhere to the relation specified in the MR. The effectiveness of MR is determined based on the mutation score (number of killed mutants / total number of mutants).

The main objective of this study is to test the NN based N-IDSs in a non-deterministic environment. To show whether the results are statistically significant, we analyzed the results over multiple iterations. The results for Application#1 (shown in Table IV) and Application#3 (shown in Table VI) are obtained based on 200 trained models for each MR (100 models trained on source data and 100 trained on follow-up data). However, due to time resource constraints, for Application#2, we obtained the results based on 60 trained models for each MR (30 models trained on source data and 30 trained on follow-up data), as shown in Table V. The results presented show the effectiveness of each MR using the mutation score (as a %). We present the mutation scores at two levels, (i) of each individual MR, and (ii) of each statistical measure (with combined MRs).

[2]https://github.com/matifkhattak/StatisticalMT/tree/master

```
total_loss = loss_weight * output_loss

total_loss = loss_weight / output_loss
```

Fig. 1. Original Code (top) and Mutant (below). This mutant will cause the program to over-fit.

It is important to note that all three applications under investigation are critical systems of high consequence, so a Type-II error has more severity because the acceptance of a *false Null hypothesis* (in case of using low α level) will suggest that there is no bug in the system when actually there is. To perform better analysis and support decision-making, we reported the results with both $\alpha = 0.05$ and $\alpha = 0.1$. Results presented in Table IV, V, and VI show that most of the time using the proposed statistical measures, the given MRs are able to kill at least 66% of the mutants. Further, if we have enough resources to apply all four statistical measures, the results (under column name *Overall*) show that the proposed approach is able to kill 100% of mutants in two N-IDSs (Application#1 and Application#2), and 66% of mutants in the cancer prediction system (Application#3), which shows its strong capability to detect defects. **Therefore, we find that the proposed statistical-based MT technique is effective in finding the implementation bugs in NN-based applications in a non-deterministic environment, which answers RQ1**.

However, it can be seen that the proposed MRs have different fault detection capability for each application under test. For example, upon closer inspection of the results in Table V, when statistically analyzed using the *t-Test*, we can see that MR-1 has a strong defect detection capability of 66.66%, whereas MR-2 has the lowest mutant killing rate of 0%. We can use the same knowledge to check the effectiveness of different MRs for different applications, as shown in Table IV, V, and VI. The results in Table VI (for Application#3) show that mutant#2 is survived and none of the MRs are able to kill it (for both α = 0.05 and α = 0.1). Upon further analysis, we find that each time a DNN is trained, it predicts the same class label for all the test instances, which shows that there is some potential bug in the classifier under test. **Hence, we come to the conclusion that different MRs have different fault detection ability for the NN-based applications under test, which answers RQ2**.

TABLE IV
RESULTS FOR APPLICATION#1: SHALLOW NEURAL NETWORK BASED N-IDS, Y DENOTES THE MUTANT IS KILLED

Significance Level (α) = 0.05													
Mutants	**Maximum Voting**			**$\tilde{\chi}^2$ / Fisher Exact Test**			**t-Test**			**Permutation Test**			**Overall**
	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	
Mutant#1	Y			Y			Y			Y			Y
Mutant#2					Y			Y			Y		Y
Mutant#3		Y	Y										Y
Mutation Score	**33.33%**	**33.33%**	**33.33%**	**33.33%**	**33.33%**	**0%**	**33.33%**	**33.33%**	**0%**	**33.33%**	**33.33%**	**0%**	
	66.66%			**66.66%**			**66.66%**			**66.66%**			**100%**
Significance Level (α) = 0.1													
Mutants	**Maximum Voting**			**$\tilde{\chi}^2$ / Fisher Exact Test**			**t-Test**			**Permutation Test**			**Overall**
	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	
Mutant#1	Y			Y			Y			Y			Y
Mutant#2				Y	Y			Y		Y	Y		Y
Mutant#3		Y	Y		Y								Y
Mutation Score	**33.33%**	**33.33%**	**33.33%**	**66.66%**	**66.66%**	**0%**	**33.33%**	**33.33%**	**0%**	**66.66%**	**33.33%**	**0%**	
	66.66%			**100%**			**66.66%**			**66.66%**			**100%**

TABLE V
RESULTS FOR APPLICATION#2: DEEP NEURAL NETWORK BASED N-IDS, Y DENOTES THE MUTANT IS KILLED

Significance Level (α) = 0.05													
Mutants	**Maximum Voting**			**$\tilde{\chi}^2$ / Fisher Exact Test**			**t-Test**			**Permutation Test**			**Overall**
	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	
Mutant#1	Y			Y			Y		Y	Y		Y	Y
Mutant#2							Y			Y			Y
Mutant#3		Y	Y										Y
Mutation Score	**33.33%**	**33.33%**	**33.33%**	**33.33%**	**0%**	**0%**	**66.66%**	**0%**	**33.33%**	**66.66%**	**0%**	**33.33%**	
	66.66%			**33.33%**			**66.66%**			**66.66%**			**100%**
Significance Level (α) = 0.1													
Mutants	**Maximum Voting**			**$\tilde{\chi}^2$ / Fisher Exact Test**			**t-Test**			**Permutation Test**			**Overall**
	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	
Mutant#1	Y			Y		Y	Y		Y	Y		Y	Y
Mutant#2							Y			Y			Y
Mutant#3		Y	Y		Y	Y							Y
Mutation Score	**33.33%**	**33.33%**	**33.33%**	**33.33%**	**33.33%**	**66.66%**	**66.66%**	**0%**	**33.33%**	**66.66%**	**0%**	**33.33%**	
	66.66%			**66.66%**			**66.66%**			**66.66%**			**100%**

TABLE VI
RESULTS FOR APPLICATION#3: DEEP NEURAL NETWORK BASED CANCER PREDICTION SYSTEM, Y DENOTES THE MUTANT IS KILLED

Significance Level (α) = 0.05													
Mutants	**Maximum Voting**			**$\tilde{\chi}^2$ / Fisher Exact Test**			**t-Test**			**Permutation Test**			**Overall**
	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	
Mutant#1	Y	Y	Y	Y			Y			Y			Y
Mutant#2													
Mutant#3	Y			Y	Y		Y	Y		Y			Y
Mutation Score	**66.66%**	**33.33%**	**33.33%**	**66.66%**	**33.33%**	**0%**	**66.66%**	**33.33%**	**0%**	**66.66%**	**0%**	**0%**	
	66.66%			**66.66%**			**66.66%**			**66.66%**			**66.66%**
Significance Level (α) = 0.1													
Mutants	**Maximum Voting**			**$\tilde{\chi}^2$ / Fisher Exact Test**			**t-Test**			**Permutation Test**			**Overall**
	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	MR-1	MR-2	MR-3	
Mutant#1	Y	Y	Y	Y			Y			Y			Y
Mutant#2													
Mutant#3	Y			Y	Y	Y	Y	Y	Y	Y			Y
Mutation Score	**66.66%**	**33.33%**	**33.33%**	**66.66%**	**33.33%**	**33.33%**	**66.66%**	**33.33%**	**33.33%**	**66.66%**	**0%**	**0%**	
	66.66%			**66.66%**			**66.66%**			**66.66%**			**66.66%**

V. Conclusion

We proposed a statistical-based Metamorphic Testing (MT) technique for testing a class of Machine Learning (ML) applications (i.e, network intrusion detection systems) that have stochastic behaviour in their results. Having this property, traditional MT approaches are not applicable because they rely on using an equality operator to verify the relation specified in Metamorphic Relations (MRs). We introduce three MRs for uncovering implementation bugs in the applications under test. Furthermore, we propose four statistical measures that allows us to statistically analyze the predicted outputs and to verify the relation specified in MRs. The effectiveness of our proposed method is show with the identification of bugs (injected through mutation testing technique) in three ML-based applications. This research is focused on showing the applicability of statistical-based MT techniques with sample MRs for neural network-based network intrusion detection systems. Furthermore, we also show the usefulness of the proposed approach in the healthcare domain (e.g., Cancer identification system). A more comprehensive study on formulating new MRs and evaluating their performance using the proposed approach is in progress.

References

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

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

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

[4] Thung, F., Wang, S., Lo, D., & Jiang, L. (2012, November). An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering* (pp. 271-280). IEEE.

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

[6] ISO/IEC 25010:2011 Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models, Mar. 2011

[7] 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*.

[8] Xie, X., Ho, J. W., Murphy, C., Kaiser, G., Xu, B., & Chen, T. Y. (2011). Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4), 544-558.

[9] Zhou, Z. Q., & Sun, L. (2019). Metamorphic testing of driverless cars. *Communications of the ACM*, 62(3), 61-67.

[10] 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).

[11] 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.

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

[13] Witten, I. H., & Frank, E. (2002). Data mining: practical machine learning tools and techniques with Java implementations. *Acm Sigmod Record,* 31(1), 76-77.

[14] Pei, K., Cao, Y., Yang, J., & Jana, S. (2017, October). Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles* (pp. 1-18).

[15] John H. McDonald http://www.biostathandbook.com/small.html, last accessed: 15 Feb 2021

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

[17] Python Software Foundation. 2013. MutPy 0.4.0. https://pypi.python.org/pypi/ MutPy/0.4.0. [Online; accessed 15-Jan-2018].