SeC-GAN: Generative Adversarial Network for Just-in-Time Defect Prediction
Shir Cohen, Amir Elmishali, Meir Kalech

To cite this version:

HAL Id: hal-03773711
https://hal.science/hal-03773711
Submitted on 9 Sep 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
SEC-GAN: Generative Adversarial Network for Just-in-Time Defect Prediction

Shir Cohen and Amir Elmishali and Meir Kalech
Ben-Gurion University of the Negev, Beer-Sheva
{shir0,amirelm,kalech}@post.bgu.ac.il

Abstract

Just-in-Time (JIT) defect prediction models identify commits that potentially induce defects in the code. When a developer writes a new commit, JIT defect prediction models are typically used to notify the developer whether the commit might induce a defect. If a commit is classified as an inducing defect, the developer is required to change the code such that it is no longer classified as an inducing defect. Developers may make merely semantic preserving changes to the commit (rather than modifying the logic), allowing the classification model to label the change as not inducing a defect. We call such a change semantic preserving changes (e.g., converting "for" loop to a "while" loop). This paper’s objective is to introduce SEC-GAN, a novel architecture based on a generative adversarial network (GAN). The architecture is designed to be sensitive to semantic preserving changes. The proposed approach is evaluated on 23 Java open-source systems. Experiments show that SEC-GAN is sensitive to semantic preserving changes rather than previous JIT defect prediction models. In particular, compared to previous Random-Forest, Logistic Regression and Fully Connected Neural Network based algorithms, with semantic preserving change samples, it succeeds in improving their F1-score by a significant increase of dozens of percent. Furthermore, even with regular samples that do not involve semantic preserving changes, SEC-GAN succeeds to compete or even to improve the prediction accuracy of previous JIT defect prediction models.

1 Introduction

People’s daily lives and workplaces have become increasingly reliant on software platforms. Consequently, the number and complexity of software systems have grown. Therefore, software defects, which can cause a cascade of unanticipated negative consequences, are inevitable [1]. Given the difficulty of developing software products without defects, suggesting appropriate approaches for identifying defects sooner is critical. The main approach is JIT defect prediction. JIT defect prediction aims to identify whether a modification in the code is expected to induce a defect. A modification can be represented by a specific file modified in the developer’s commit. The purpose of JIT defect prediction is to help developers identify defects early during the code writing phase and deliver better quality products.

JIT defect prediction is a classification problem. Previous studies addressed this problem, based on various commit information, such as commit message [2] or static analysis warnings [3]. Machine learning techniques have been widely used in existing work for building JIT defect prediction models [4][5]. Recent works [2][5][6] used deep learning techniques to improve the performance of JIT defect prediction models.

JIT defect prediction models are typically used to warn a developer when inserting a new commit whether the modification may induce a defect to the file. In case the developer’s modification is classified as introducing defects, she is expected to change the code so that the amended modification is classified as not introducing defects. Nevertheless, many references in the literature confirm that developers tend to ignore the model’s classifications [7]. For example, [8] analyzed 280 Stack Overflow questions regarding static analysis tools that can be used to identify potential code defects. They found that the second most prevalent question was "how validation of false positives". The developer assumed that the analysis tool classified the code she wrote that might be defective but thought that the code she wrote did not induce defects. Therefore, she examined ways to make the tool not mark the code as an inducing defect. One way to do this is to make a semantic preserving change only. A semantic preserving change will be defined in the paper as a change that does not affect the program’s logic behavior. In other words, it does not affect functionality. Converting a "for" loop to a "while" loop is an example of a semantic preserving change. While such a modification does not affect the program’s logic behavior, it may cause the classifier to change from the classification of inducing a defect to not inducing a defect. Indeed, a developer can change the model’s classification in this way. It is worth noticing that 13 of the 36 questions examined on the Stack Overflow website were eventually classified as causes of the defect (true positive), although the programmer thought otherwise. So, the programmer’s attempt to hide the defect classification may induce a defect.

This paper presents two main contributions: (1) identifying the weaknesses of traditional JIT algorithms as a result of semantic preserving changes, (2) presenting a new architecture, named SEC-GAN (Semantic Preserving Changes Generative Adversarial Networks), that aims to be sensitive to semantic preserving changes and improve pre-
vious JIT defect prediction models. SEC-GAN is based on generative models that have become increasingly relevant and popular in various disciplines (e.g., for music generation [9]). Generative models are used to generate adversarial instances in order to achieve neural network robustness [10]. In particular, a generative model generates instances that reflect modifications similar to the data set but challenging to classify. In this study, we use this technique to develop a novel architecture that generates new instances helping the model be sensitive to semantic preserving changes. **SEC-GAN offers two benefits:** (1) It is more sensitive to semantic preserving changes than previous JIT defect prediction methods. (2) It is not restricted to dealing only with semantic preserving changes, but it suggests a model that performs at least as well as previous JIT defect prediction models on regular instances.

## 2 Related Work

JIT defect prediction involves two tasks, first generating a training set, then training a prediction model. Previous studies proposed methods for both research questions. Next, we present a literature review for each challenge.

### 2.1 Training Set Generation

The feature extraction for generating a training set is a challenging task, and many previous papers addressed them.

**Feature Extraction**

Trautschet et al. [3] use a defect prediction method based on static source code metrics. They found that static source code and static analysis warning features improve the prediction quality of JIT defect prediction models.

Kamei et al. [4] utilized static code analysis metrics to analyze whether a change has induced a defect in the source code. They selected 14 factors classified into five dimensions: diffusion, size, purpose, history, and experience. Many papers have utilized these metrics. As an example, McIntosh and Kamei [11] utilized these metrics and included code review metrics for the predictive models.

Hoang et al. [2] proposed a deep learning framework, DeepJIT, that automatically extracts features from commit messages and code changes and uses them to detect JIT defects. In addition, they used static code analysis features from McIntosh and Kamei [11].

In this paper, we use static source code metrics as proposed by Trautschet et al. [3]. These metrics analyze a program’s source code without running it, for example, Logical Lines of Code.

### 2.2 Classification Algorithms for JIT Defect Prediction

Many classification algorithms have been used for generating a JIT defect prediction models. Some research [3; 4] use a logistic regression model to perform predictions. Trautsch et al. [3] propose an additional model, Random Forest, for JIT defect prediction task.

The field of JIT defect prediction has recently shown an increase of interest using deep learning techniques. Ferenc et al. [12] revitalized static code analysis metrics and combined them with deep learning to flag suspicious code segments at the class level. They implemented a deep neural network and compared to machine learning models such as KNN, random forest and logistic regression. They found that deep learning and random forest perform the best compared to the other classifiers in terms of F1-score. They mention that more data is likely to increase the performance of their approach even better.

Deep learning is used to extract features as well [6; 13]. Wang et al. [6] learn the semantic representations of programs automatically from source code files and code changes with a deep learning approach. They used token vectors taken from the applications’ abstract syntax trees (AST) to train a deep belief network. Also, Li et al. [13] used AST and build convolutions neural networks (CNN) to classify the defect-prone source code. Hoang et al. [2], like Li [13], used CNN to extract features that represent commits’ meaning from commit messages and code changes.

Our research uses a GAN to generate new instances and thus improve the model and cope with semantic preserving changes in the code. To the best of our knowledge, no previous research has incorporated generative models to improve the model sensitivity to semantic preserving changes.

## 3 Problem Definition and Methodology

This section formally defines the JIT Defect Prediction problem and shows how to use machine learning classification algorithms to construct a JIT defect prediction model.

### 3.1 Problem Definition

Typically, projects are stored in repositories in the modern software development process. To maintain the repository, a version control system keeps track of the changes in the software components, named commits. The granularity level of the software component that contains a modification could be in different levels such as package, file, class or method. Our research defines an instance at the file level (JIT-F). A commit consists of several modifications that each modify accomplished in a specific file.

JIT defect prediction is described as the task of predicting which modifications in the repository are expected to induce a defect in the code. JIT defect prediction can be modeled as a classification problem. Formally, given a software project with $n$ commits $C = \{c_1, \ldots, c_n\}$. Each commit is composed of a set of modifications $c_k = \{m_{k1}, \ldots, m_{ks}\}$ so that $q$ represents the number of modified files in commit $k$: the task is to determine the state of a modification $m_{ks}$ - whether or not the modification induced defect to file $i$.

Supervised machine learning algorithms are commonly used to solve classification problems. They obtain as input a set of labeled instances. A labeled instance is a pair consisting of a set of features that describe the instance and its label. In our case, an instance includes features that describe the modification $m_{ks}$, and its label is True if $m_{ks}$ induces a defect or False otherwise. The labeled instances are collected to a training set used to train a classification model. Next, we describe how to generate a dataset and use classification models for JIT defect prediction.

### 3.2 Methodology

To generate a training set, we first explain how we extract the features and label each instance.

**Features:** Many features were proposed in the literature for JIT defect prediction models. Following previous studies [2; 3; 4; 5], we utilize static source code metrics as features. Static code analysis is the process of analyzing the source
code of a program without executing the program. For example, Nesting Level Else-If.

**Label:** In order to determine whether a modification induced a defect, we use a variant of the SZZ algorithm [14]. It proceeds as follows: First, we use the BEIRUT tool [15] to identify the modifications that fixed bugs. Then, for each modification that fixed the bug, we identify which previous modification induced that bug. To this end, we use PyDriller [16]. In particular, given the modification that fixed the bug, PyDriller returns the set of modifications that last "touched" the lines that are modified in the files included in the modification. With this approach, we can link a defect (label) reported in the issue tracker to the modification that induced it.

**Classifiers:** Given a training set, a machine or deep learning classifier is used to generate a model that predicts whether a new modification will induce a defect in the code. A key factor in the success of prediction models is choosing a suitable classification algorithm. Many possible models were proposed in the literature for JIT defect prediction [4]. In the [Evaluation chapter] we describe the main methods we compare, which can be used with the static features. In this case, a user can easily alter the classifier by choosing a modification that the prediction model will classify as one that does not induce a defect, even if it does in fact, [17]. Next, we explain this weakness in detail.

**Semantic Preserving Changes**

We first define the meaning of a semantic preserving change. Then we show how a semantic preserving change can be used to alter the prediction of a modification.

**Definition 1 (Semantically Equivalent Code).** Let $\text{out}(f, x)$ be a function that returns the output of source code $f$ given an input $x \in \text{Input}$. Let $f_1$ and $f_2$ be two source code segments. We say that $f_1$ and $f_2$ are semantically equivalent code, if $\text{out}(f_1, x) = \text{out}(f_2, x)$ for every $x \in \text{Input}$.

**Definition 2 (semantic preserving change).** Given a change in code segment $f_1$. Let $f_2$ represents the modified code segment. We say that this change is a semantic preserving change if:

- (i) $f_1 \neq f_2$
- (ii) $f_1$ and $f_2$ are semantically equivalent code.

According to Definition 1, as long as we make semantic preserving changes to a source code $f_1$, the modified source code $f_2$ and the original code $f_1$ are semantically equivalent code. For example, Figure 1a illustrates a code that is modified with semantic preserving changes. The initial code is in part (a), whereas parts (b)-(d) illustrate three code segments state that $X$ equals 5. Then, if $X$ is an even number, increase it by 1. Otherwise, increase it by 2.

As previously stated, most JIT prediction methods use static source code metrics as features. These metrics are directly affected by the semantic preserving changes in the code; however, the label of the modified semantically equivalent code segments remains the same. Figure 1b illustrates a few static source code metrics for each code segment in Figure 1a. The semantic preserving changes, as shown, affect the static metrics. However, we still expect the classifier to predict the same label for each code segment since the code segments are semantically equivalent code.
that represents a modification in the code. The input of the classifier can be derived from two sources: (1) real data that is extracted from a version control system, and (2) fake data - the output instances of GB and GNB. The output of the classifier is the class of the modification - inducing defect or not (c).

The classifier is based on models from the literature that have solved the problem of JIT defect prediction, as we will expand in the Evaluation section. These models use as tabular input data representing static source code features.

It should be noted that generating data with semantic preserving changes is difficult due to the various rules that might be applied. For example, in Figure 1a, it is feasible to execute three alternative rules to get semantically equivalent code segments. To deal with an enormous number of rules, we built the generator (GB and GNB) using the idea of Generative Adversarial Networks (GAN) [10]. Generative models can generate new instances that are indistinguishable from real data and fit within the input variable’s distribution. As a result, some of the generated instances can represent as semantically equivalent code segments that are correlated to real data. These instances are used to train the classification algorithm, which improves its sensitivity to semantic preserving changes.

Standard GANs consist of two neural networks, a generator G and discriminator D. Informally, these two networks are in competition – G tries to generate samples that resemble real data, while D attempts to discriminate between samples generated by the generator and real data samples. GAN has been used in a variety of applications, including image synthesis tasks [18] and natural language processing [19].

Static source code metrics, used for JIT defect prediction, however, are typically organized as tabular data, which is more challenging for GAN architectures. To cope with these challenges, we built our architecture using Conditional Tabular GAN (CTGAN) [20]. CTGAN is a synthetic tabular data generator based on conditional GAN.

The training process includes two steps:

**Step 1** - We train the generators GB and GNB based on CTGAN architecture.

**Step 2** - Next, we train the Sec-GAN architecture as presented in Figure 2, where the training set includes the modifications’ features and their labels. The training process proceeds as follows. First, extract real instances from the project repository, Xreal. Second, generate fake instances with GB and GNB models, in the same size as the training set, Xfake. Specifically, GB generates fake samples based on the number of defects in the real data, whereas GNB generates fake samples based on the number of not defects in the real data. Then, Train a JIT defect prediction model with the training set, Xreal + Xfake. In the Evaluation section we thoroughly explain the parameters of the JIT defect prediction model.

To summarize, the architecture of Sec-GAN allows the construction of a sensitive model. This is because the model has been trained with additional instances that it may incorrectly classify, giving it greater sensitivity to classify the same label to semantically equivalent code instances. On the other hand, traditional models from the literature, do not use generative models, and are hence less sensitive to semantic preserving changes, and as a result, these instances may result in prediction mistakes.

### 4 Evaluation

#### 4.1 Experimental Settings

**Datasets:**
We evaluate the performance of Sec-GAN, we generated datasets that include instances that describe modifications in a file’s granularity level, as well as a label for each instance indicating whether the modification induced a defect. The datasets include 23 Java open-source projects of the Apache Software Foundation.

**Feature Extraction:**
Static source code metrics are commonly used as features for JIT defect prediction [2, 4, 5]. We use some of those features based on static features that have been shown to be useful in previous research [3, 21]. In particular, we use four groups of metrics, as follows:

- **Change-based features:** We applied javadiff [24] to calculate the diff metrics that consider the size of the change. For example, number of lines added, (2) **Abstract Syntax Trees (AST) difference:** To determine fine-grained modifications, we measure the differences between the ASTs before and after a change. To extract these features we use the GunTree tool [25]. (3) **Statement-level difference:** We use javalang [26] implementation to extract the modified code statements of a modification. For example, number of while statements, (4) **Static code metrics:** The static source code metrics include object oriented metrics as well as size and complexity metrics collected by OpenStaticAnalyzer [27]. For example, number of public methods.

**Data Pre-processing:**
We cleaned the extracted data to reduce outliers and noise as follows. Filter out commits that changed at least 100 files. Then, filter out modifications that changed at least 10,000 lines and modifications that did not change source lines since they cannot induce a defect. Following, filter out modification that deleted files or renamed them and modifications that changed test file. Finally, filter out modifications with value of nan and remove duplicate instances.
This process is based on concepts given in the research of [23] and included only modifications that changed files that end in "_.JAVA".

To standardize the metrics’ values, we decreased each value by its mean and divided by its standard deviation, resulting in a Gaussian distribution.

Our datasets are quite imbalanced, i.e., the number of defect-inducing modifications represents only a tiny percentage of the total changes. To mitigate the class imbalance in our data, we performed SMOTE sampling. The SMOTE was performed only for the training set to evaluate the algorithms fairly.

**Competitive JIT Prediction Models:**

One main objective of the experiments is to evaluate the sensitivity of the JIT prediction model to semantic preserving changes. Semantic preserving changes directly affect static source code metrics. As a result, we compared SEC-GAN to known algorithms in the literature that rely on tabular data with static source code metrics as input. In particular, we compare SEC-GAN to the following JIT defect prediction methods:

- **Random Forest (RF)** - based on the method described in [3; 2] and implemented with sklearn. To fine tune RF, we run GridsearchCV that finds the number of trees by optimizing the F1-score.
- **Logistic Regression (LR) -** as presented by [3] and implemented with sklearn. To fine tune LR, we also run GridsearchCV that finds the penalty, solver and l1 ratio by optimizing the F1-score.
- **Fully Connected Neural Network (FCNN) -** we performed experiments to find the best neural network for our data. The network is built based on the architecture described in [12], with some improvements optimizing it on our datasets and implemented with PyTorch. In particular, we built a FCNN deep learning model. It consists of one input layer, two hidden layers and one output layer. The number of neurons in each fully connected layer is 128 and 256, respectively. We added a dropout layer at a rate of 0.3 after each fully connected layer. We used a $\tanh$ activation function after each fully connected layer and $\text{Sigmoid}$ for the output layer. We trained with the Adam optimizer and set the learning rate of real example model $\alpha = 2 \times 10^{-3}$ and batch size 200. To train the model with the fake samples, we also trained with Adam optimizer and set $\alpha = 2 \times 10^{-2}$ and batch size 30. We trained the model with 200 epochs. The objective function is Binary Cross Entropy (BCE).

There are additional deep learning models for JIT defect prediction suggested in the literature that we have not compared to. Most of them, [13; 2], do not use static analysis features and instead use the raw data of the commits as input to the deep learning models, making them incomparable to our approach.

To implement SEC-GAN architecture, we used the default parameters of CTGAN as defined in the implementation of CTGAN[7]. The training set produced by CTGAN for each project can be found here[7]. For each project, the fake data used to train the models (RF, LR and FCNN) was the same.

The models were generated by the above algorithms using 80% of the data for training, and 10% of the data for testing the models. From the training, we assigned 10% for validation to perform tuning. The data has been stratified, thus the percentage of bugs in the training, validation and test sets is equal.

**Semantically Equivalent code Generation:**

To simulate the process by which a programmer can change the model classification from defect-inducing to non-defect-inducing. It demonstrates how a developer can use semantic preserving changes to alter the model prediction. To that end, we performed an evaluation of the model on the test set ($X_{test}$). After executing a JIT fault prediction model with $X_{test}$, $X_{test}$ can be divided into four groups: (1) $TP$ (true positive) is a set of the defect-inducing instances correctly predicted by the model, (2) $FP$ (false positive) is a set of non-defect-inducing instances that were incorrectly predicted as defect-inducing, (3) $TN$ (true negative) is a set of non-defect-inducing instances correctly predicted as such, (4) $FN$ (false negative) is a set of defect-inducing instances incorrectly predicted as non-defect-inducing.

The ECG procedure attempts to manipulate TP instances by changing them as similar to existing negative instances as possible (FN or TN). This approach enables the creation of instances that induce defects but may cause the classification to be misled. The ECG procedure creates semantic preserving changes to the TP instances by operating predefined rules, some of which are inspired by [24]. The full set of rules, called Semantic Change Rules, can be found here[7].

It is worth to note that we constructed a naive semantic preserving changes algorithm by defining semantic preserving changes patterns. The semantic preserving changes we performed exploit the use characteristics of the machine-learning model to demonstrate the significant influence that semantic preserving changes may have on the model. At the same time, the rules we have set are basic and few, so we do not consider them to be part of the research innovation or methodology. These rules could be generalized in future work by considering additional patterns.

The procedure of generating similar code is described in Algorithms 4. As previously stated, the ECG attempts to modify $tp \in T$ as similar as possible to an existing negative instance (FN or TN). In line 6, we check the similar-
```plaintext
Algorithm 1: Semantically Equivalent Code Generation

Input: JIT defect prediction model - E
Input: The feature set - F
Input: A test set - X_{test} = TP ∪ FP ∪ TN ∪ FN
Result: A new test set with semantic preserving changes - X_{test changes}

1. N ← TN ∪ FN
2. for tp ∈ TP do
   3. tp′ ← tp
      4. while true do
         5. can_change ← False
         6. N_modifiable ← sort N based on their similarity to tp′ (using Euclidean distance)
         7. for n ∈ N_modifiable do
            8. tp_change ← Semantic_Change(tp′, n)
            9. if tp_change! = None then
               10. can_change! = True
               11. tp′ = tp_change
               12. break
            end
         end
         13. if can_change == False then
            14. break
         end
   15. TP ← TP ∪ {tp} ∪ {tp′}
16. end
17. X_{test changes} ← TP ∪ FP ∪ N
18. return X_{test changes}
```

For each rule, the algorithm checks to verify that all of the change instances. The Semantic_Change procedure succeeds in manipulating tp in such a way that it is similar to n. If the change can be applied, tp changes, so we will recalculate the similarity of the new tp to N and repeat the process. Otherwise, we move on to the next negative example. The process will end when tp cannot be changed for each n ∈ N. The new instance then exchanges tp in the group TP (line 18). The algorithm returns the new test set X_{test changes}, which includes the semantic preserving change instances. The Semantic_Change procedure go over each rule. For each is checked to verify that all of the features affected by the rule are indeed changed in tp to resemble n, as well as the preconditions of the rule are fulfilled. If tp is changed we will return the instance after the change. Otherwise, we will return None.

To summarize, the algorithm gets the test set X_{test} (evaluated according to a specific classifier) and returns a new test set X_{test changes} after changing the instances classified as TP. Consequently, the algorithm adjusts to a particular classifier’s weaknesses since it considers the model’s N samples and so allows for maximum vulnerability in that model.

Research Questions:

Our research goal is to present S2C-GAN, a novel approach for JIT defect prediction that is sensitive to semantic preserving changes that result in semantically equivalent code. To evaluate the contribution of S2C-GAN instances, we address the following research questions:

- **RQ1.** How sensitive is S2C-GAN approach to semantic preserving changes compared to other state-of-the-art JIT prediction models?
- **RQ2.** Is S2C-GAN approach competitive to other state-of-the-art JIT prediction models?

To address RQ1 and RQ2, we compare each one of the previously mentioned approaches (in Section 4) RF, LR, FCNN), trained with two sets of instances:

1. X_{real}, denoted as classifier, where classifier ∈ {RF, FCNN, LR} (e.g. RF).
2. X_{real} ⊇ X_{fake}, denoted as S2C-GAN(classifier), where classifier ∈ {RF, FCNN, LR} (e.g. S2C-GAN(RF)).

To address RQ1, we compare these two variations using a test set that only contains instances belonging to X_{test changes} set generated by the ECG process. To address RQ2, we compare these two variations using a test set that only contains instances belonging to X_{real} (instances generated from real projects commits).

**Evaluation Metrics:**

We use common metrics from the literature [3,4] to evaluate the prediction performance: precision, recall and F1-score. In addition, we compute a relative percentage between our S2C-GAN architecture and the competitive JIT defect prediction models as follows:

$$Increase(\%) = \left(\frac{metric_{S2C-GAN(classifier)} - metric_{classifier}}{metric_{classifier}}\right) \times 100$$

where metric∈ {F1-score, precision, recall}, classifier∈ {RF, FCNN, LR} refers to the classifier trained with instances from real projects (X_{real}), and S2C-GAN(classifier) refers to the classifier additionally trained with instances generated by S2C-GAN architecture (X_{real} ⊇ X_{fake}). The average increase, used as a measure in the results, was calculated by averaging the Increase metric across the projects. The fraction is undefined when the metric value for a specific classifier is zero. In this case, if metric_{S2C-GAN(classifier)} = 0 we set Increase to zero too. Otherwise, we did not include Increase for that project when averaging across the projects.

To check if our results are statistically significant, we use Wilcoxon signed-rank test [25] and Cliff’s δ to further analyze the experimental results.

**4.2 Results**

This section discusses the obtained results, focusing on the research questions we presented above.

To address RQ1, we applied the semantically equivalent code generation (ECG) process to generate a new test set X_{test changes}. Table 1 shows the results of the prediction models. The three left columns present the F1-score, precision, and recall averages. Then the average percentage increase of each previously defined metric is presented. Recall that the ECG procedure might result in a division by 0, we computed the average increase across all the projects with division by no zero. The percentage of projects in which there was a division by zero are 0%, 30% and 18% while using RF, LR and FCNN, respectively.

The results in Table 1 demonstrate that a S2C-GAN is more sensitive to semantic preserving changes. The most significant improvement is seen when the classifier is
FCNN, where F1-score grows from 0.08 to 0.17 and provides an average increase of almost 185%. Also, the F1-score of LR show a major improvement, from 0.09 to 0.15. A smaller improvement, from 0.12 to 0.13, can be seen in the F1-score of RF. Since the ECG process attempts to change the classification of instances from TP to FN, we can see its effect on the recall metric. In this metric, we can see an average increase of 11.26%, 85.53% and 267.20% while using RF, LR and FCNN, respectively.

It is worth noting that all models produce a lower F1-score for the test set than in earlier research. This is not surprising given that $X_{test\_changes}$ comprises examples that are difficult to classify since, despite being positive, their features are similar to those of negative samples. In datasets similar to ours, the F1-score reported in the literature is $\sim 0.25$ [3]. As seen below, the models achieve the same F1-score as the literature on the $X_{test}$.

Finally, we apply Wilcoxon signed-rank test and Cliff’s $\delta$ to determine whether the difference between each classifier and SeC-GAN is statistically significant. If and only if the p-value is less than 0.05 and Cliff’s $\delta$ is greater than or equal to 0.147, the performance of SeC-GAN is significantly different from the classifier; otherwise, the difference is negligible. We have bolded in Table 1 the values that are statistically significant different.

To address RQ2, we evaluate the algorithms using the test set, $X_{test}$. In this experiment we would like to verify that the great achievement of SeC-GAN in semantic preserving changes test set, does not have a negative impact on regular instances ($X_{test}$). Table 2 presents the results in the same format as in Table 1. It shows that SeC-GAN architecture succeeds in achieving F1 scores that are comparable or even better than previous JIT defect prediction models. Specifically, when the classifier is FCNN, the F1-score grows from 0.25 to 0.27, an average increase of almost 12%. LR achieves a smaller improvement, and for RF there is no improvement in terms of F1-score. Note that all models produce nearly identical predictions, between 0.25 to 0.27 in terms of F1-score. These results are consistent with those in the literature in datasets similar to ours [3].

Also in this experiment we apply wilcoxon signed-rank test and Cliff’s $\delta$ and we have bolded in Table 2 the values that have statistically significant differences ($p_{value} < 0.05$ AND $|\delta| >= 0.147$).

### 5 Threats to Validity

This section discusses the threats to validity of our work.

**Construct Validity:** We adopted the evaluate metrics as they have been widely utilized in prior prediction tasks studies in general [26], and specifically for JIT defect prediction [13] [12].

**Internal Validity:** The process of labeling defect-inducing commits is a threat to validity of the generalization of our results. However, to deal with the SSZ limitations, we rely on the benchmark generation tool [25] to get all bug fixing commits ,and we used PyDriller’s SSZ implementation [16].

**External Validity:** We evaluated our proposed approach on 23 Java open-source projects of the Apache Software Foundation. Apart from the fact that these projects are used in many studies in the field [3] [5], they may easily be extended to other programming languages and software projects to make SeC-GAN more general. Another threat that can be considered is the comparison to other methods in the field of JIT defect prediction [13] [2]. The above comparison is not possible due to the limits of the ECG algorithm. At the same time, we compare our method to the RF method, which outperformed several other models in the field [12]. In addition, further work can be extended to the ECG to fit these models.

**Conclusion Validity:** To validate the results, we compared previously proposed machine learning and deep learning models. We chose RF and LR for machine learning algorithms since they have been shown to be effective in a number of previous studies [3] [12]. Also, the deep learning models showed good results in state-of-the-art works [5] [2]. In this study, we tuned several training parameters to achieve the optimum performance of each algorithm.

### 6 Conclusions

This paper dealt with JIT defect prediction. We showed that existing JIT defect prediction models have weakness that they are not sensitive to semantic preserving changes that generate semantically equivalent code. These changes may cause the prediction model’s classification to be modified, while the defect-inducing remains. Because this weakness, we suggest that training any JIT model with semantic examples is necessary. Furthermore, we advocate expanding the technique to additional domains.

We proposed a new architecture, SeC-GAN, that includes a generative model. As a result of the training process, the classifier could be trained with more instances that are difficult to classify. Evaluation on 23 Java open-source projects demonstrated that SeC-GAN is more sensitive to semantic preserving changes than state-of-art work while maintaining prediction quality. Specifically, SeC-GAN increases the F1-score measure by 9.83%, 110.67% and 184.6%, compared to RF, LR and FCNN, respectively. In addition, SeC-GAN succeeds to compete or even to improve RF, FCNN and LR models, on instances that do not include semantic preserving changes.

### References


