# IntJect: Vulnerability Intent Bug Seeding

Benjamin Petit[1,2], Ahmed Khanfir[2], Ezekiel Soremekun[2], Gilles Perrouin[1], and Mike Papadakis[2]
[1]University of Namur, Namur, Belgium
[2]University of Luxembourg, Luxembourg, Luxembourg
petitbenj@gmail.com, ahmed.khanfir@uni.lu, ezekiel.soremekun@uni.lu,
gilles.perrouin@unamur.be, mike.papadakis@gmail.com

***Abstract*—Studying and exposing software vulnerabilities is important to ensure software security, safety, and reliability. Software engineers often inject vulnerabilities into their programs to test the reliability of their test suites, vulnerability detectors, and security measures. However, state-of-the-art vulnerability injection methods only capture code syntax/patterns, they do not learn the intent of the vulnerability and are limited to the syntax of the original dataset. To address this challenge, we propose the first intent-based vulnerability injection method that learns both the program syntax and vulnerability intent. Our approach applies a combination of NLP methods and semantic-preserving program mutations (at the bytecode level) to inject code vulnerabilities. Given a dataset of known vulnerabilities (containing benign and vulnerable code pairs), our approach proceeds by employing semantic-preserving program mutations to transform the existing dataset to semantically similar code. Then, it learns the intent of the vulnerability via neural machine translation (Seq2Seq) models. The key insight is to employ Seq2Seq to learn the intent (context) of the vulnerable code in a manner that is agnostic of the specific program instance. We evaluate the performance of our approach using 1275 vulnerabilities belonging to five (5) CWEs from the Juliet test suite. We examine the effectiveness of our approach in producing compilable and vulnerable code. Our results show that INTJECT is effective, almost all (99%) of the code produced by our approach is vulnerable and compilable. We also demonstrate that the vulnerable programs generated by INTJECT are semantically similar to the withheld original vulnerable code. Finally, we show that our mutation-based data transformation approach outperforms its alternatives, namely data obfuscation and using the original data.**

***Index Terms*—Software Vulnerabilities; Vulnerability injection; Software Security; Software Reliability***

## I. INTRODUCTION

Fault seeding is a popular technique for assessing the bug-finding capabilities of test generators [1], [2] and assessing test suite thoroughness [3]. It is often employed to guarantee that specific types of faults are not present in the software under test [2], [4] and to form test requirements that can judge the adequacy of testing [5]. The underlying idea is to seed bugs that encode the characteristics of the targeted faults and check the ability of bug-finding processes to uncover them.

Fault seeding usually involves syntactic changing the programs under test to inject faults that conform to the targeted fault type. Typically, faults are seeded based on syntactic patterns [5] that change benign code into buggy code. While effective, such an approach usually results in trivial faults [3] and numerous false positives, i.e., faults not belonging to the targeted fault class. This issue has the unfortunate effect of producing misleading test assessments or bug-finding measurements as it obscures the computed scores. Trivial faults result in inflated measurements [3] while false positives result in deflated measurements and wasted resources [6].

Additionally, the design of security-aware fault injection methods has received particularly low attention. Most previous works aim to inject logical faults [5]. Hence, it remains challenging to inject security-related faults for security testing and test assessments. To deal with this issue, *we propose a vulnerability injection approach (called* INTJECT*) that aims at injecting specific types of security-related bugs*.

Some researchers have proposed security-related fault seeding approaches such as Lava [1] and Pitest [7]. However, these approaches are limited to few predefined patterns and they make working assumptions that inhibit their general purpose application and extension. For instance, Lava [1] injects vulnerabilities by introducing guard conditions that reflect the path conditions of a given test case. This means that the resulting vulnerable codes are dependent on the provided test cases and the path conditions, they do not capture the vulnerability context or the type of the intended bugs. Similarly, Pitest [7] injects blindly a number of supported fault patterns/types without any consideration of the fault context. More importantly, both approaches require significant efforts to be applicable to other languages and vulnerability types.

This work addresses these challenges by proposing the first intent-based vulnerability injection method (INTJECT) which seeds specific types of security-related bugs by learning their context from given examples. Automatically learning from examples is an important direction of research, especially considering the plethora of new vulnerabilities reported over time. Thus, we propose INTJECT– a learning-based method that effectively learns both the context of the code (e.g., where vulnerability is to be injected) and the syntactic transformation required to inject these vulnerabilities.

This is particularly challenging because learning-based methods are data-hungry, they require rich and large datasets to be effective. However, such vulnerable datasets are unavailable or scarce in practice [8]. To deal with this issue, INTJECT effectively learns from few examples using a combination of NLP (natural language processing) methods and semantic-preserving program mutations (at the bytecode level). This combination allows to effectively learn both the vulnerability intent (i.e. context) and the required syntactic transformation from benign to vulnerable code.

Specifically, given a set of known vulnerabilities (containing benign and vulnerable code pairs), INTJECT automatically infers their patterns by repeatedly mutating the vulnerable

```
1  String stringNumber = readerBuffered.readLine();
2  data = Integer.parseInt(stringNumber.trim());
3  - if (data < array.length) // vulnerable code
```

```
3  + if (data >= 0 && data < array.length) // benign code
```

```
4  {
5      int a, b = 10; // sample mutation (addLocalMutation)
6      IO.writeLine(array[data]);
7      a = 15; // sample mutation (addLocalMutation)
8  }
```

(a) Program showing vulnerable, benign and mutated code snippets

***Intent***: *inject CWE-129 into benign code*

***Syntactic Difference***: *Difference on line 3: "`data >= 0 &&`"*

***Vulnerability Context***: *Lines 1 to 8 (N <= 100 tokens surrounding the syntactic difference (line 3))*

***Semantic-preserving Mutation***: *Line 5 & 7 E.g., "`int a, b = 10;`" (addLocalMutation)*

(b) Features of Vulnerable Code

Fig. 1: Motivating Example showing a program containing a vulnerability (CWE 129), and the the vulnerability context learned by INTJECT to enable effective vulnerability injection in benign code

and clean code examples using *semantic-preserving mutations*. These mutations aim to ensure that INTJECT learns the context of the vulnerable code in a manner that is general and applicable to other programs, other than the initial examples.

The vulnerability intent is learned through a neural machine translation (i.e., Seq2Seq) model. The main idea is to learn the vulnerability context (e.g., vulnerability-related variables, conditions and methods). It is important to capture vulnerability context since it allows to select tokens within a context and reason about the vulnerable code transformation [9].

We evaluate the performance of our approach using 1,275 vulnerabilities belonging to five (5) CWEs provided by the Juliet test suite [10]. We examine if INTJECT *effectively* injects vulnerabilities in benign code by checking if the resulting code is valid (compilable) and vulnerable. We then analyse the contribution of our semantic-preserving mutations to the effectiveness of our method. To show that our model indeed captures the vulnerability intent, we investigate the semantic similarity of the vulnerable code generated by INTJECT to the original vulnerable code in the dataset. Figure 1 presents an example of vulnerability intent learned by INTJECT.

Our results provide empirical evidence that INTJECT is effective in generating syntactically valid yet vulnerable code, almost all (99%) of the code produced by INTJECT is vulnerable and compilable. Furthermore, INTJECT injects vulnerabilities that are semantically similar to real vulnerable code. This shows its utility for evaluating vulnerability-detection tools. Finally, we observe that our mutation-based data transformation approach contributes to the performance of INTJECT. It outperforms the state-of-the-art data sources, i.e., *only* data obfuscation and *only* the original data. This result encourages the adoption of INTJECT's mutation technique to augment data for similar code learning-based applications.

The rest of the paper is organized as follows: Section II presents an overview of our approach and Section III discusses our methodology. Section IV presents our experimental setup, and Section V reports our results. Sections VI and VII discuss threats to validity and related work. Finally, Section VIII concludes the paper.

## II. OVERVIEW

This section illustrates the challenges with learning-based vulnerability injection with an example and demonstrates how our approach (INTJECT) addresses this problem.

### A. Motivating Example

Consider a program (shown in Figure 1a) containing a well-known vulnerability (CWE-129), namely the improper validation of an array index obtained from the program input. To be valid, the array index must: 1) be greater than or equal to zero and 2) be strictly less than the array length ("benign" line 3). The absence of the first conditional check 1) creates a vulnerability (see "vulnerable" line 3, in Figure 1a), allowing for negative indices to be sought for in the array. This may lead to illegal memory access and buffer underflow. Indeed, an attacker can exploit this vulnerability to craft memory-related attacks, e.g., resource leakage (*see* CWE-839).

### B. Intent of Vulnerability Injection

Given a pair of benign and vulnerable code samples (e.g., Figure 1a), the *goal* of vulnerability injection is to successfully seed the vulnerability captured in the vulnerable code into an *unseen* benign code. We posit that to successfully inject such a vulnerability into an arbitrary benign code, it is important to learn the *intent of the vulnerability*. This intent is composed of the *program syntax* and the *vulnerability context*, i.e., the code (e.g., variables and methods) surrounding the vulnerability.

*1) Syntactic difference:* In this motivating example, the *program syntax* is captured by the *syntactic difference between the benign code and vulnerable code*. It is important to learn the type of code snippets to add, delete or modify to successfully inject a vulnerability into an arbitrary benign code. However, this is not sufficient since inserting the learned syntactic difference or pattern in any random `if` condition does not necessarily introduce the vulnerability. Specifically, a successful injection requires knowledge of the context of the vulnerability, i.e., the code sequences around the vulnerable code snippet.

*2) Vulnerability Context:* The *vulnerability context* is important to understand the code location and the conditions under which the vulnerability is effective. For instance, injecting the missing check into any other `if` condition in the program does not introduce the vulnerability. It has to be an `if` condition before the use of the array index, and the vulnerable code location has to be followed by a read operation from an external source. In our motivating example (Figure 1), the vulnerability context refers to the code sequences surrounding the vulnerable line(s) of code (lines one to eight). The context size principally depends on the NLP approach used and is set empirically (see Section IV).

### C. Limitations of the State-of-the-art

In this section, we discuss the limitations of existing state-of-the-art vulnerability injection methods.

*1) Limited Syntactic Patterns:* Loise *et al.* [7] inspect manually vulnerable-benign (fixed) code pairs to derive security-aware mutation operators. By manually analysing the Find-Bugs [11] database, the authors created 15 mutation operators for injecting vulnerable mutants. These operators capture the vulnerability intents for each studied vulnerability. This work is limited by the fact that it *manually* captures vulnerability intents. Meanwhile, INTJECT automatically learns the intents.

Similarly, BUG-INJECTOR inserts vulnerabilities using predefined templates [12]. Unlike mutation techniques, Bug-injector determines the location of the injection by running test cases of the program to be altered by bugs and matching the execution information with templates preconditions. The vulnerability intent is predefined rather than learnt as INTJECT does. In contrast, IBIR exploits bug reports to identify source code locations to be made faulty and inverts program repair fixes to inject faults [13]. Notably, both BUG-INJECTOR and IBIR do not specifically target vulnerabilities.

*2) Dataset Limitations:* Learning-based vulnerability injection methods rely on learning vulnerability patterns from an existing dataset. Thus, the capability of the learned model is often limited to the program features available in the original dataset, meaning that changes in such features in an arbitrary program could reduce its performance. Hence, the original dataset influences the effectiveness of the vulnerability injection model. Moreover, the low variety of syntactic features as well as little context for the vulnerability reduces the injection accuracy. We posit that successful vulnerability injection in arbitrary code requires learning several vulnerability contexts and a varying set of program contexts. Hence, it is important to transform the original datasets with diverse program contexts that preserve the program semantics and vulnerability intent.

To address these dataset concerns, we propose a mutation-based data manipulation of the original dataset. Our data manipulation approach performs semantic-preserving program mutations on the original dataset to increase the diversity of program features as well as the diversity of program context surrounding the vulnerability. For instance, consider the motivating example (Figure 1), INTJECT mutated the original code to add a new variable initialization (in lines 5 and 7). This mutation preserves the intent of the vulnerability, while increasing the diversity of the program context.

### D. INTJECT

The goal of this work is to learn both the *program syntax* and *vulnerability context* to enable the successful injection of seen vulnerabilities in *unseen* benign code. In our motivating example (Figure 1), INTJECT learns both the syntactic difference capturing the vulnerable statement (in line 3), and the vulnerability context embodied by the statements around the vulnerable code. Learning the combination of the vulnerability context and the syntactic difference is an effective gadget to insert vulnerabilities into arbitrary programs. Figure 2 illustrates how INTJECT works. It firsts transforms the original dataset into a mutated dataset via semantic-preserving program mutations. These mutations increase the diversity of the program syntax and varies the context of the vulnerability while preserving the program semantics and the vulnerability intent. Then, our sequence to sequence (seq2seq) transformer model learns both the program syntax and vulnerability intent. Our evaluation shows that INTJECT is effective in vulnerability injection because it learns both program syntax and vulnerability context from a diverse set of programs (Section V).

## III. METHODOLOGY

Let us describe how INTJECT creates vulnerable programs from benign ones. INTJECT learns the vulnerability intent and program syntax via a combination of mutation-based data transformation and neural machine translation (NMT). It first transforms the training data using semantic-preserving mutations, then learns a vulnerability injection model via a seq2seq model [9]. Figure 2 depicts INTJECT's workflow.

In the following, we describe in detail the workflow of our approach (INTJECT) in two main steps:

- **Step 1 - Semantic-preserving Program Transformations:** Given a dataset containing pairs of benign and vulnerable code, our approach (INTJECT) transforms the dataset via semantic-preserving mutations. In particular, it generates several instances of the code pairs that preserve the program semantics and the vulnerability. To achieve this, INTJECT applies bytecode level mutation to transform both the vulnerable code and the benign code into semantically similar, but syntactically different equivalents. It first compiles each benign/vulnerable program, then perform code transformations in the bytecode version of the program such that it does not break the vulnerable code semantics. We perform this step using a modified version of CONFUZZION [14] (*see* Section IV-C). INTJECT then decompiles the resulting mutated code pair into a new benign/vulnerable source code pair. This source code has a similar behavior as the original one, but it is syntactically different due to the mutation and decompilation steps. INTJECT also ensures that the transformations preserve the presence or absence of the vulnerability in the code using INFER [15]. The resulting *mutated* dataset allows our approach (INTJECT) to learn

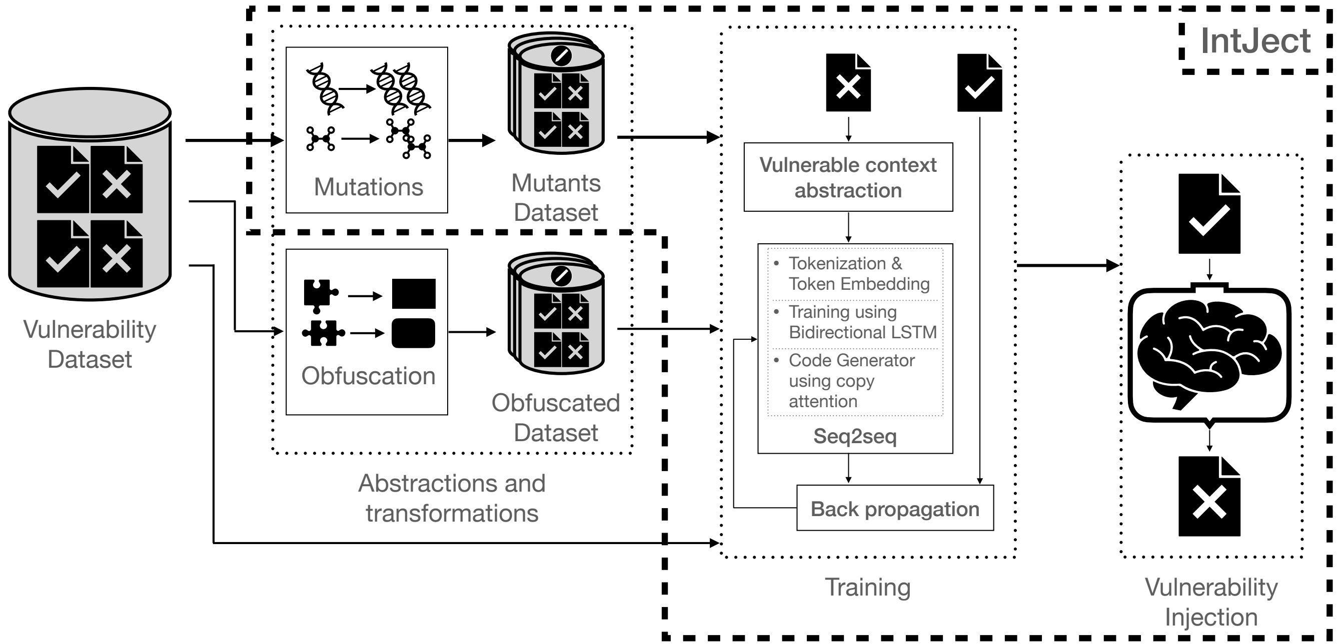


Fig. 2: Our experimental workflow showing INTJECT's phases the mutation-based program mutation of INTJECT, and alternative data transformation approaches (i.e., obfuscated dataset and original vulnerability dataset)

the vulnerability intent under different circumstances, and discriminate it from the program syntax.
In addition, our evaluation involved examining the contribution of our mutation-based data transformation, in comparison to the state-of-the-art data transformation approaches (*see* RQ2 section V). These approaches include using the original raw dataset and using an *obfuscated* version of the dataset (Figure 2). Generally, *obfuscation* [16] attempts to protect code from being reverse-engineered, without modifying its functionalities. Precisely, it replaces all components given-names (like parameters, variables, etc.) by arbitrary names, thus, making code hard to read by unwanted users (i.e. attackers). In this work, we compare *default* INTJECT (i.e., using the mutated dataset) versus using an obfuscated dataset. Obfuscation is applied on each pair of vulnerable and benign code to yield new obfuscated pairs. Typically, the goal of such obfuscation-based data transformation is to prevent the model from associating irrelevant information with vulnerabilities and guide its learning towards their behavioral aspects. Obfuscation in this work was performed using the "Java Obfuscator" provided by "Semantic Designs" [17]. For both obfuscated and mutated pairs, we ensure the presence of vulnerabilities by relying on the INFER static analyzer [15]. Overall, we generated two additional datasets resulting from the aforementioned obfuscation and mutation operations, respectively.

- **Step 2 - Neural Machine Translation (NMT):** For this task, INTJECT starts by 1) performing vulnerable context abstraction, then 2) it splits the input (benign-vulnerable code pairs) into token-sequences, next 3) it feeds them to the Seq2Seq learning architecture (using a Bidirectional LSTM model, and global copy attention [9]) to train the model towards injecting vulnerabilities in benign code. The employed learning architecture uses supervised learning and back-propagation to learn the weights for inference/prediction. INTJECT training phase is based on a Sequence-to-Sequence (Seq2Seq) transformer model architecture popularly employed in learning-based language translation [18]. It employs a learning approach similar to that proposed by Chen et al. [9] which was designed to capture fault context for bug-fixing, i.e., automated program repair (APR). In this work, we re-purpose this architecture for vulnerability intent and injection, i.e., security-related fault seeding. Figure 2 shows the details of our NMT training phase. In the training phase, we have adapted the hyper-parameters to avoid over-fitting to the input dataset and to increase the prediction performance of the model, particularly for unseen code (*see* Sections IV-F and IV-F). After model training, the outcome of our training step is a Seq2seq model which generates vulnerable code snippets, and inject vulnerabilities. In the inference mode, given a benign code, INTJECT uses the token generator of the resulting model to generate the vulnerable version of the benign code.

## IV. EXPERIMENTAL SETUP

In this section, we discuss our research questions (RQs) and provide details about our experimental setup.

### A. Research Questions

We design and conduct experiments to evaluate the effectiveness of INTJECT (RQ1) and the contribution of our semantic-preserving mutations (RQ2). We also investigate the semantic similarity of the vulnerable code generated by INTJECT to the vulnerable code in the original dataset (RQ3). Specifically, we ask the following research questions (RQs):

- **RQ1 Effectiveness:** How effective is our approach (INTJECT) in generating valid and vulnerable code?
- **RQ2 Semantic-Preserving Program Mutations:** What is the contribution of our mutation-based data transformation approach to the effectiveness of INTJECT? How does it compare to alternative datasets, i.e., using the original dataset or an obfuscated dataset?
- **RQ3 Semantic Similarity to Original Vulnerable Code:** How semantically similar are the vulnerable code generated by INTJECT to the original vulnerable code (from the dataset), in terms of program dependence?

### B. Vulnerability Dataset (CWEs) and Oracle

*1) Vulnerability Dataset:* In this work, we use the Juliet Test Suite (version 1.3) [19]. This vulnerability dataset was created by the Center for Assured Software (CAS) of the National Security Agency (NSA) to assess the capabilities of static analysis tools to detect some vulnerabilities coming from the Common Weakness Enumeration (CWE) – a collection of known software security issues [20]. It contains well-known security issues in software systems, including 11 of the 2011 CWE/SANS 25 topmost dangerous software errors. Each CWE contains "test cases" that demonstrate specific vulnerability (or CWE). These test cases are created using a "Test case Template Engine" tool or manually injected into different benign source code snippets. In this paper, we employ 1275 vulnerabilities containing five (5) CWEs from the Juliet Dataset. We have chosen these five CWEs and the Juliet dataset due to their high prevalence in typical software systems and popularity in the research community, notably in vulnerability detection [21] and vulnerability analysis [22]. Table I provides details about our vulnerability dataset. Our dataset contains 4060 test cases, i.e., Java programs containing at least one vulnerable method and several benign versions of the method. In our setting, a test case refers to a Java program from the Juliet dataset containing at least one vulnerable method and several (two to four) benign methods. From each test case, we obtain several pairs of vulnerable and benign code.

*2) Oracle:* We used INFER [15], a static analyser developed by Facebook, to determine the presence or absence of a vulnerability in a program. INFER detects security vulnerabilities in Java or C (C++, Objective-C) programs via static program analysis. We validate that our test oracle (INFER) works correctly via a preliminary evaluation on the vulnerability dataset. We test that for all code pairs in our dataset, INFER (a) does not miss-classify a benign code as vulnerable, (b) correctly identifies a vulnerable code snippet, and (c) detects the specific vulnerability (CWE) in the vulnerable code.

TABLE I: Details of Experimental Datasets

| CWEs | Description | # instances |
|---|---|---|
| **129** | Improper validation of array index | 155 |
| **134** | Uncontrolled format string | 62 |
| **190** | Integer overflow | 499 |
| **191** | Integer underflow | 499 |
| **369** | Divide by zero | 60 |
| **Raw Data** | Original dataset | 1275 |
| **Obfuscated Data** | Obfuscation Data Manipulation | 1275 |
| **Mutated Data** | Mutation-based Data Manipulation | 2227 |

### C. Experimental Datasets

In our experiments, we train INTJECT on three different datasets, namely the (1) original dataset from the Juliet dataset, (2) an obfuscated dataset and (3) the *default* mutated dataset produced by our semantic-preserving program mutations. In the following, we describe each dataset:

*1) Original Dataset:* Table I provides details on the data preparation of the original (raw) dataset. This is the original vulnerability dataset which can be directly used for training without any data transformation or abstraction. For this dataset, we obtained 6991 pairs from 4060 test cases. We then filtered for pairs with the token restriction (<=150 tokens) resulting in 1347 instances. We also removed duplicated or identical benign methods to finally obtain 1275 program pairs.

*2) Obfuscated Dataset:* In this work, we employ obfuscation to abstract the code syntax while retaining the semantics. This is to create a dataset that is more general than the original raw data since it abstracts away syntactic sugar. We compare the performance of our approach using the obfuscated data to that of our semantic-preserving program mutations. To obfuscate the original dataset, we use the tool "Java Obfuscator" developed by "Semantic Designs" [17].

*3) Mutated Datasets:* In this work, we propose the use of semantic-preserving program mutations to transform an original dataset into a rich and diverse one that allows to learn the vulnerability intent under varying contexts. To mutate the original dataset, we employ CONFUZZION [14]. Basically, CONFUZZION is a mutation-based feedback-guided black-box JVM fuzzer, which is focused on the type confusion vulnerabilities detection. In our case, the interesting component of this tool is the mutation module. CONFUZZION supports several program mutations at the method and class level. However, for this work, we employ only the semantic preserving mutations at the method-level detailed in Table II. To this end, we adapt the tool to our needs, mainly by disabling some mutation operators, targeting the input code instead of the JVM classes, and incorporating it to the pre-processing phase of INTJECT. Then, we run it on our dataset, generating 50 mutants per vulnerable code. Thus, ending up with 50 mutated vulnerable codes for every target code pair. To generate the benign version of the vulnerable mutated versions, we applied the same changes (caused by the mutation) on the original benign version. Finally, we ensure that these mutations are semantic-preserving for each pair of vulnerable and benign code by

TABLE II: The Mutation Operators used from Confuzzion (adapted from [14])

| Category | Mutation operator | Description |
|---|---|---|
| Method | AddLocalMutation | adds a new local variable and generates a corresponding object. |
| | AssignMutation | assigns an existing value to a new variable. |
| | CallMethodMutation | adds a method call and generates the necessary arguments. |

TABLE III: Data Preparation of the mutants.

| Filter / Dataset | Raw | Mutated |
|---|---|---|
| **# Test cases** | 4060 | 21487 |
| **# Instances** | 6991 | 39464 |
| **# Instances (# tokens** <= 150) | 1347 | 15007 |
| **# Instances unique** | 1275 | 2227 |

using INFER; where we check that the vulnerability is present in the vulnerable code and absent in the benign code.

Table III shows the details of the mutated datasets in comparison to the original dataset, in terms of the number of test cases and resulting program pairs ("instances"). Our mutation-based data transformation produced 75% more instances after performing 50 mutations per test case. We also remove the equivalent mutants resulting from this step.

### D. Research Protocol

1) *Data Preparation and Pre-processing:* First, we pre-process the original dataset from the Juliet Test Suite [19] to obtain the pairs of vulnerable and benign code. This dataset was analysed, filtered and prepared to obtain the code pairs applicable for our settings. For instance, we filtered for code instances that are no more than 150 tokens due to token length limitations of employed learning frameworks. This is in line with settings found in related works [9], [23].
2) *Preliminary Experiments with Test Oracle:* We then feed all code pairs from our dataset to INFER for analysis to ensure that all vulnerable code can be detected and correctly classified. We also ensure that benign codes are not misclassified. In fact, we conduct all the experiments with instances that are correctly classified by INFER. We refer to the resulting dataset from this step as the *original (raw) dataset*.
3) *Data Transformations:* Next, we perform the two data transformations on the dataset, namely data obfuscation and semantic-preserving program mutations. These transformations resulted in two more datasets, which we call respectively the *obfuscated dataset* and the *mutated dataset*.
4) *Model Training:* We perform a preliminary hyper-parameter setting and tuning (reported in subsection IV-F). We then train INTJECT on each dataset using the Seq2seq library in OpenNMT [18] for fixing bugs – SEQUENCER [9]. We call the resulting model INTJECT with mutated dataset, original (raw) dataset or Obfuscated dataset. We also train under different model configurations (i.e., 2,000, 6,000 and 10,000 model training iterations). All reported results (e.g., Table IV) are over a five-fold cross-validation.
5) *Experimental Data Analysis:* In our analysis, we assess the performance of our approach (INTJECT) to inject vulnerabilities into benign code snippets. In particular, using metrics described in subsection IV-E we analyse the effectiveness of INTJECT for the different datasets and model configurations (RQ1). We also analyse the performance of our semantic-preserving mutations (RQ2) and the semantic similarity between the generated code by INTJECT and the original vulnerable code (RQ3).

### E. Metrics and Measures

*1) **Predictive Accuracy**:* We compute the following metrics for all experiments, resulting INTJECT models, different model configurations (2000, 6000 and 10000 iterations) and datasets (original, obfuscated and mutated datasets):

- *Ratio of compilable predictions:* This is the proportion of programs generated by INTJECT that are *syntactically valid* code, i.e., could be compiled by the Java Compiler.
- *Ratio of vulnerability inserted:* This refers to the proportion of programs generated by INTJECT that were determined to be vulnerable by INFER.

*2) **Semantic Difference** (SOOTDIFF):* To determine the semantic similarity between the vulnerable code generated by INTJECT and the original vulnerable code, we use the difference at the bytecode level to abstract away the syntactic differences. To this end, we used SOOTDIFF [24], a tool designed to identify if two bytecodes are from the same source code. It allows to determine if the bytecode representation of the original vulnerable code matches that of the code generated by INTJECT. Consequently, a match of both bytecodes allows to determine that they are semantically similar, that is, they represent the same vulnerable program behaviour. A difference allows to measure the degree of mismatch at the bytecode level, which shows that even if the vulnerability is present in both code snippets (as determined by INFER) there is a degree of behavioral difference between both codes. We have selected SOOTDIFF for our computations since it is more reliable than bytecode or syntactic code clone detection. Besides, it shows behavioral differences (e.g., differences in program dependencies), and abstracts away syntactic differences.

*3) **Levenshtein Distance**:* We use the popular levenshtein metric [25] as a semantic distance metric to measure the difference reported by SOOTDIFF. To compute this, we report the Levenshtein distance between all differing instructions between the two code snippets, i.e., the original vulnerable code in the dataset and the code generated by INTJECT.

### F. Training Configurations & Settings

***Main Hyper-parameters:*** We set the seed size to one (1) for all experiments to mitigate randomness and allow to replay/compare all experiments in a balanced setting. Our model training involves two (2) encoder and decoder layers, with a Bridge

layer between the last encoder state and the first decoder state. The models are trained using an RNN architecture, in particular an LSTM [26]. We also employ global attention and copy mechanisms to improve the capabilities of the model.

***Preliminary Experiments:*** First, we conduct a preliminary experiment to determine the hyper-parameter setting for our approach, using 24 manipulated instances containing five (5) CWEs from the Juliet dataset. The goal of this experiment is to perform a parameter sweep of variables under varying circumstances to determine stable parameter values across board and avoid over-fitting. In this experiment, we also train with the mutated and the obfuscated test datasets to determine the appropriate parameters. Experimenting with these different transformed datasets helps to ensure that the parameters are not over-fitted to a specific dataset and the approach works.

***Evaluation Metrics:*** For each parameter setting, we evaluate the performance of each model by computing the *accuracy* of the model based on the statistics provided in the OPEN-NMT framework[1][2]. We compute the accuracy as follows: $Accuracy = 100 * \frac{number_of_correct}{number_of_words}$. In addition, we conduct a manual analysis of the predictions to understand how the model reacts to each parameter modification. This allows to determine the stable parameter setting for our experiments.

***Hyper-parameter Tuning:*** In a preliminary experimental phase, we examine combinations of the following hyper-parameters: *earlystopper*, *GRU versus LSTM*, *Three versus two layers*, *100 vs. 250 vs. 500 vocabulary sizes*, *16 vs. 32 vs. 64 batch sizes*, *100 vs. 256 vs. 512 word vector sizes*, *absence vs. presence of the bridge layer* and *100, 256 and 500 RNN sizes*. From this investigation, we determine the best-performing parameter setting to be used in our experiments, which is the following: training with an RNN of size 256 (same as in Tufano et al. approach [27]), using a batch size of 32 and Word vec size of 256 (similarly to Sequencer [9]) and a vocabulary size of 250, which fits within the range of sizes used by other notable NMT approaches; between 129 and 1000 vocabulary sizes [9], [28], [29]. Our model is trained using two layers for the encoder and the decoder model, in particular, we employ a bidirectional LSTM for the encoder and a simple LSTM for the decoder. Our token generator also employs attention and copy mechanisms.

***Model Training and Testing:*** Overall, INTJECT was trained on our mutated dataset containing 2227 instances. In addition, we trained two models for comparison as baselines, one model was trained on the raw dataset and a second on the obfuscated datasets. Both the raw and obfuscated datasets contain 1275 instances each. For all trained models, we conducted a five-fold cross-validation, with an 80%-20% train-test split.

### *G. Implementation Details and Platform*

Our approach (INTJECT) and experimental analysis were implemented in over 2KLoC (2281 LoC) of Python code.[3] Our implementation is built on top of the OPENNMT framework[4] in Pytorch on a DELL computer with an Intel(R) Core i7 2720qm running Windows OS equipped with a Debian virtual machine (VM). All our prototypes are single-threaded. Experiments were conducted on Google Collab[5] (free edition) with 13 GB of RAM, 40 GB of Hard drive, CPU Intel(R) Xeon(R) CPU @ 2.30GHz, GPU Tesla T4.

[1] https://github.com/OpenNMT/OpenNMT-py/blob/7c314f41dc1b017ac105144beeb53cb072960a54/onmt/utils/statistics.py

[2] https://opennmt.net/OpenNMT/training/logs/

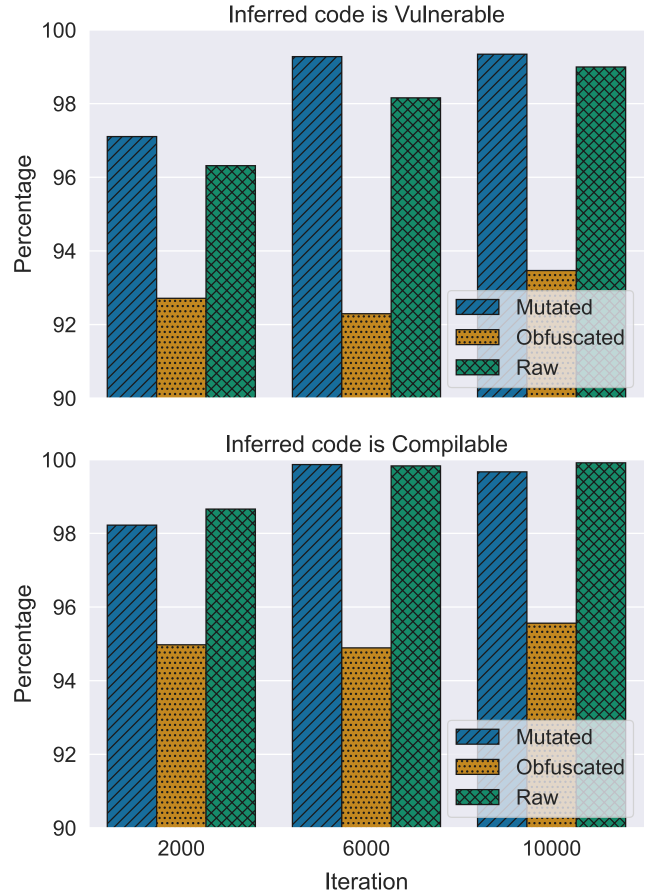


Fig. 3: Performance (Accuracy) of INTJECT at different model configurations (training iterations) for different datasets (original raw, obfuscated and mutated datasets)

## V. RESULTS

In this section, we evaluate the effectiveness of INTJECT in vulnerability injection. We also examine the contribution of our mutation-based data manipulation approach and the semantic similarity of the vulnerable code generated by INTJECT to the original vulnerable code.

[3] https://github.com/Petit-Benjamin/VulnerabilityInjectionNLP

[4] https://opennmt.net/

[5] https://colab.research.google.com/signup

TABLE IV: Vulnerability Injection Effectiveness of INTJECT (using 10000 model training iterations and $< 100$ tokens). Best performance are in **bold text**. ("#" = "Number of", "%" ="Percent of", "Obf." = "Obfuscated", "Impr." = "Improvement")

| Generated Code | INTJECT with | | | | | | % Impr. over Raw Data INTJECT using | | % Impr. over Obf. Data INTJECT using | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Default (Mutated) | | Raw Data | | Obfuscated Data | | | | | |
| | # | % | # | % | # | % | Obf. Data | Mut. Data | Raw Data | Mut. Data |
| #Compilable | 1512 | 99.67% | 1193 | **99.92%** | 1141 | 95.56% | -4.36% | **-0.25%** | **4.56%** | 4.30% |
| #Vulnerable | 1510 | **99.34%** | 1182 | 98.99% | 1116 | 93.47% | -5.58% | **0.35%** | 5.91% | **6.28%** |

### A. RQ1 Effectiveness

In this experiment, we investigate the effectiveness of our approach (INTJECT) in successfully injecting a vulnerability into benign code. Specifically, given benign code, we examine the effectiveness of INTJECT in transforming the benign code into vulnerable code. This experiment employs the *default* setting of INTJECT, i.e., using the dataset produced by our mutation-based data manipulation approach. Figure 3 and Table IV ("Default (Mutated)" column) provide details of the effectiveness of INTJECT under this setting.

We found that *our approach (*INTJECT*) is effective in injecting security vulnerabilities into benign code.* Almost all (99.67% of) programs produced by INTJECT are compilable, and most (99.34% of) programs produced by INTJECT are both *vulnerable* and *compilable* (*see* Table IV "Default (Mutated)" column). This result suggests that our approach effectively learns both the intent of the vulnerable code as well as the program syntax required to inject these vulnerabilities. Figure 3 further shows that *default* INTJECT (with mutated data) is effective (96 to 99%) across different model training configurations (i.e., 2,000, 6,000, and 10,000 training iterations). However, we observed that INTJECT performs best at higher training iterations (6,000 and 10,000). Overall, INTJECT is highly effective (up to 99% accuracy) in vulnerability injection across datasets, albeit its default mutated data mode consistently outperforms the alternative data manipulation approaches (raw and obfuscated data). We believe INTJECT is effective in vulnerability injection because it accurately learns the intent of the vulnerability, i.e., the context of the vulnerability as well as the program features required to effectively inject these vulnerabilities.

> INTJECT *is highly effective in injecting software vulnerabilities into benign code: Almost all (99.34% of) programs produced by* INTJECT *are vulnerable and valid.*

### B. RQ2 Semantic-preserving Program Mutation

This experiment examines the contribution of our semantic-preserving mutation-based data transformation approach to the effectiveness of INTJECT. To address this, we evaluate how INTJECT performs given the three different datasets, namely the original raw dataset, an obfuscated dataset and a mutated dataset produced by our data transformation approach. In particular, we compare the performance of INTJECT when using mutation-based transformation approach versus the original raw dataset or the obfuscated dataset. Figure 3 and Table IV ("% Impr. over Raw/Obf. Data INTJECT using Mut. Data") highlight the results of this experiment.

Our results show that INTJECT *is (up to 6.28%) more effective in vulnerability injection than the original dataset or the obfuscated dataset*. Specifically, Table IV and Figure 3 show that mutation-based data transformation improves the accuracy of INTJECT over the original dataset and the obfuscated dataset by 0.35% and 6.28%, respectively. Inspecting the number of generated compilable code, INTJECT (with the mutated dataset) is better than the obfuscated dataset but comparable to the original raw dataset. In comparison to the obfuscated dataset, our mutation-based data transformation improves the percentage of compilable code by 4.30%. We also observed that this performance is comparable to that of the original dataset, it is only slightly (-0.25%) less effective than the original dataset. The performance of our mutation-based data transformation suggests that it is comparable to the original dataset, but more effective than the obfuscated dataset. This is due to the semantic-preserving nature of our program mutations and its ability to increase the diversity of the vulnerability context.

> *The mutation-based data transformation of* INTJECT *is (6.28%) more effective in vulnerability injection than the obfuscated dataset and comparable to the raw dataset.*

### C. RQ3 Semantic Similarity to Original Vulnerable Code

In this experiment, we compare the semantic similarity of the vulnerable code generated by INTJECT to the original vulnerable code in our dataset. For each generated code pairs, we examine if the vulnerable code generated by INTJECT is semantically similar to the original vulnerable code in the vulnerability dataset by comparing the difference in the bytecode representation of both programs, using SootDiff [24]. This gives the difference in the program dependence between both programs while abstracting from syntactic sugar such as specific variable names. For a balanced evaluation, we do not account for the distance between `String` values because of the obfuscated dataset to avoid inflated computation of semantic differences. We report the semantic difference in terms of the Levenshtein difference between the bytecode representations of both programs. Table V and Figure 4 show the results of this experiment for each dataset and configuration.

Our evaluation results show that given benign code, INTJECT *generates vulnerable code that are highly semantically similar to the original code in the dataset*. Specifically, the semantic difference between the generated code by INTJECT

TABLE V: Percentage of compilable ("Comp.") and vulnerable ("Vuln.") programs generated by INTJECT which are semantically different (SootDiff > 0) from the original vulnerable code in the Vulnerability Dataset, for different datasets, and model configurations (number of training iterations). Best Results (i.e., lowest difference) are in **bold text**. ("%" ="Percent of", "Obf." = "Obfuscated", "Reduc." = "Reduction of Vulnerable Code", "N/A" = "Not Applicable")

| #Model Training Iterations | %Comp. Code by INTJECT | | | %Vuln. Code by INTJECT | | | % Reduc. over Raw Data INTJECT using | | % Reduc. over Obf. Data INTJECT using | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Default (Mutated) | Raw Data | Obf. Data | Default (Mutated) | Raw Data | Obf. Data | Obf. Data | Mut. Data | Raw Data | Mut. Data |
| 2000 | 4.02 | **2.63** | 3.21 | 2.83 | 0.51 | **0.44** | **-0.14** | 454.90 | **15.90** | 543.18 |
| 6000 | **0.97** | 2.91 | 1.67 | 0.26 | **0.00** | 0.44 | N/A | N/A | **-100** | -40.91 |
| 10000 | **0.33** | 2.36 | 1.09 | **0.00** | 0.25 | 0.35 | 40 | **-100** | -28.57 | **-100** |

and the original vulnerable code is very low, they share almost the same program dependence features. For instance, consider the experiments with INTJECT (mutated dataset versus raw dataset) at 10,000 model training iterations which is the best performance across all settings. We observed that only 0.33% of the compilable code generated by default INTJECT (using mutated dataset) are semantically different from the original code in the vulnerability dataset (last row, second column of Table V). This corresponds to about five (5) programs with semantic difference (SOOTDIFF Levenshtein difference) less than 50 (Figure 4). Meanwhile, 2.36% of the compilable code generated by INTJECT using the *original raw dataset* are semantically different (SootDiff > 0) from the original code in the vulnerability dataset (last row, third column of Table V), for the same setting. This corresponds to about 13 programs with Levenshtein difference up to 250 (Figure 4). Overall, Table V shows that most vulnerable code generated by INTJECT match the original vulnerable code at the default configuration with 10,000 model training iterations. This is evident by the low Levenshtein distance (of zero at 10,000 iterations) between the vulnerable code and the generated code by INTJECT (*see* Table V). These results suggest that INTJECT generates vulnerable code that are highly similar to the original vulnerable code in the vulnerability dataset.

INTJECT *automatically generates vulnerable code that are quite similar (semantically) to the original vulnerable code. This means that the program dependence difference between the generated and original vulnerable code is very low (frequently as low as zero).*

Additionally, we found that *our mutation-based data transformation has the best performing semantic similarity*. Comparing the semantic similarity of the generated code for the different datasets under different model configurations, we observed that our mutated dataset performs best, especially after 10,000 model training iterations. In this setting, INTJECT using the mutated dataset outperforms the obfuscated and original dataset by up to 100% reduction in semantic difference between vulnerable and generated code (e.g., last column, last row of Table V). This performance is followed by the raw dataset and the obfuscated dataset. Figure 4 further shows that *default* INTJECT (with the mutated data) outperforms both the original dataset and the obfuscated dataset in generating semantically similar valid code. It generates fewer

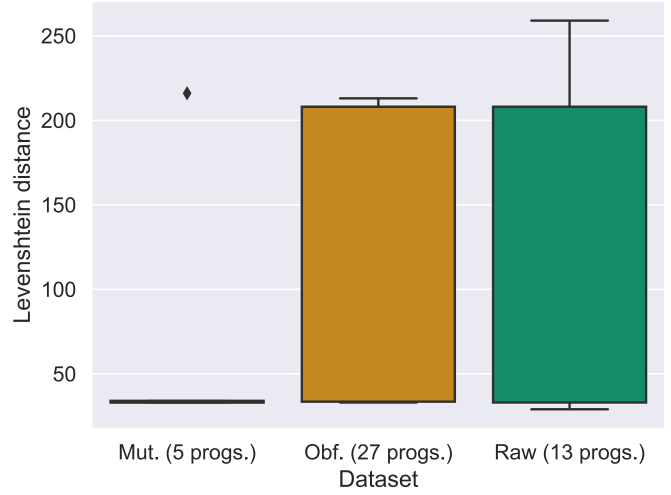


Fig. 4: Semantic difference (in SOOTDIFF Levenshtein distance) between generated valid (i.e., compilable) code and the original code for each dataset (at 10,000 iterations).

(5) valid programs that are semantically different from the original dataset and it has a very low Levenshtein distance.

Inspecting the effect of model training iterations, we observed that as the number of model training iterations increases, the number of generated programs that are semantically different (semantic difference > 0) from the original vulnerable code reduces to almost zero for the mutated dataset. At lower model iterations the number of semantically different programs is slightly higher at about 0.26% at 6,000 iterations and at the worst 2.83% at 2,000 iterations (*see* Table V). We observed that mutating the dataset outperforms the original dataset and data obfuscation. We attribute this performance to our semantic-preserving program mutations which improves the dataset diversity via varied vulnerability contexts.

INTJECT *using the mutation-based data transformation outperforms both the obfuscated dataset and the raw dataset in generating semantically similar vulnerabilities.*

## VI. THREATS TO VALIDITY

This section discusses the limitations and threats to the validity of our approach, experiments and findings.

**External Validity:** This refers to the generalizability of our approach and results. The most important threat to external validity is posed by the employed vulnerability dataset – Juliet dataset [19]. While we are certain that INTJECT and our findings hold for this dataset, there is the threat that our approach and results may not hold for other vulnerability datasets. Besides, INTJECT may not generalize to new or different vulnerabilities. To mitigate this threat we have experimented with five different CWEs from the Juliet dataset. Indeed, our results and approach hold for these vulnerabilities, but may not generalize to other vulnerabilities. However, our work demonstrates that this approach (INTJECT) is applicable to mature and up-to-date vulnerabilities (CWEs) in vulnerability datasets such as Juliet. Juliet is a well-maintained vulnerability dataset containing up-to-date CWEs reported by developers. Hence, we expect that our findings and approach are applicable to the tested CWEs and similar datasets.

**Internal Validity:** The threat to internal validity is posed by the incorrectness of our implementation, specifically, whether we have correctly implemented INTJECT for vulnerability injection (as described). We have mitigated this threat by testing our implementation of the approach and data analysis pipeline with written tests. We have also conducted a manual inspection of preliminary steps and results to ensure our implementation works as expected. For instance, to validate our data analysis pipelines, we manually inspected our code and checked whether INFER, which is independent of our dataset, finds the reported/generated vulnerabilities. This is a relatively good sanity check that we indeed inject vulnerabilities.

**Construct Validity:** The main threat to construct validity is posed by the semantic-preserving mutations. There is a potential bias that there are duplicates or semantically similar mutants in the resulting mutated datasets, especially since we mutate the original dataset before performing train-test splits. However, to mitigate this threat, we have filtered our equivalent mutants in the resulting mutated dataset.

In addition, it is possible that (mutated) code snippets considered as benign in our setup contain other vulnerabilities, i.e. unknown or undetectable by INFER. Although this may threaten the validity of studies in adjacent research directions, i.e. assessing vulnerability detection techniques, it does not represent a major threat in our scope – injecting vulnerabilities in code, be it benign or not. Yet, to mitigate this threat, we consistently use the same definition of ”vulnerable” and ”benign” for all treated codes, based on INFER predictions. Still, as we rely on INFER to detect vulnerabilities, our work may be impacted or inherit the limitations of this latter. Besides, the only definite proof of a vulnerability is to (automatically) exploit it, we consider the tasks of effective vulnerability detection and automatic vulnerability exploitation as out of the scope of this work. Moreover, automatically exploiting or detecting vulnerabilities in any arbitrary code remains an ongoing research theme in the community [4], [30].

## VII. Related Work

**Vulnerability injection:** PiTest (an extended version of PIT [31]) is a tool using syntactic transformations to seed vulnerabilities into a Java code by inverting the Findbugs patterns. The tool supports 15 mutation operators designed to insert security vulnerabilities on Java programs [7] using syntactic matches. As such it cannot automatically learn or extend this set without any manual effort and analysis as INTJECT. Moreover, PiTest does not consider the surrounding code context, where the patterns apply, restricting its ability to inject some complex types of bugs.

Lava is a tool developed to inject and validate buffer overflow vulnerabilities in C code. To do so, this tool inserts a bug by looking at situations where an input can trigger a read or write overflow [1]. The tool then introduces guard conditions, based on the trace of a test case that executes this location, such that when the conditions are met a vulnerability is considered to be discovered. In essence, this results in a case where every test traversing the trace of the test case that was used to inject the vulnerability also reveals it. Unfortunately, this method is only suitable for evaluating fuzzers as the resulting code is artificial, easily detectable by static analysis techniques, and of limited bug types. Moreover, generalizing it to other types of vulnerabilities remains a challenge. In contrast INTJECT can support multiple types of vulnerabilities and it can learn their intent and produce more natural vulnerable code.

**Data transformation:** Evilcoder [32] is a tool able to detect where to inject vulnerabilities and remove the security mechanism to make the code vulnerable, using a dataset transformation approach. It uses static analysers to find sensitive sinks matching bug patterns. This means that it only supports guard condition deletions. As this tool uses static analysis to inject vulnerabilities, its purpose is more to generate new test corpora than to evaluate bug-finding techniques.

**NLP for Fault injection (not vulnerability injection):** CodeBERT [33] is an NL-PL model designed by Microsoft researchers. This is a pre-trained model for Programming Languages. The usage of this model in the fault injection domain shows its ability to seed "natural" faults that, we can say, semantically resemble real faults [34], [35]. Effectively, the faults injected resemble what a real programmer could write (regarding the programmatic rules, convention, etc) [36].

CodeBERT has also been applied for mutation testing [35], [36]. For this purpose, researchers used the Masked Language Modeling task that takes a sentence with one masked token and the goal of the model is to find the most likely tokens to replace it. This model can take up to 512 tokens as input, it then predicts the five best solutions for the masked token as output. Being already pre-trained on more than six (6) million programs, the tool can inject faults. It creates mutants that emulate the behaviour of real faults. We note that CodeBERT is not specifically trained on a dataset containing faults, but rather on general corpus of programs to learn the "language" of code [37]. However, unlike INTJECT, this approach does

not necessarily inject vulnerabilities or security-related bugs.

Tufano et al. [27] have also used a bi-directional RNN Encoder-Decoder to automatically insert bugs in code. The dataset used to train their model is a corpora of bugs-fixes from GitHub repositories from which they extracted pairs of vulnerable-benign methods of maximum 50 tokens. Their fault-injection tool is on the method-level. In fact, one method is supposed to implement only one task and contain enough context for the model. Hence, their model can take as input a method of maximum 50 tokens and should predict the buggy version of this method. They also experiment the same work with larger methods, up to 100 tokens but with lower prediction performance (when the prediction reintroduces the original bug) (around 6% vs around 20% for the 50 max tokens version of the tool). This work is focused on injecting faults in programs with no specific focus on security-related bugs. In contrast, INTJECT is focused on injecting vulnerabilities into benign code snippets.

**NLP for Vulnerability Prediction:** Existing work in vulnerability prediction aims to define hand-crafted program features (such as libraries, function calls and code churn) that could be associated with vulnerabilities in order to detect such vulnerabilities [30], [38]–[41]. Other approaches have proposed learning-based methods to detect vulnerabilities, in particular, using NLP techniques such as text mining (bag of words) [42], [43]. Unlike the aforementioned works, in this work, we are focused on vulnerability injection rather than vulnerability detection. Our approach is able to generate vulnerable code from clean benign code using a combination of semantic-preserving program mutations and NLP-based machine learning methods – Neural Machine translation (Seq2Seq).

## VIII. Conclusion

This paper presents a vulnerability injection method (called INTJECT) that captures program syntax and vulnerability intent via a synergistic combination of semantic-preserving program mutations and neural machine translation (Seq2Seq). We demonstrate that our approach effectively learns program syntax and vulnerability intent. INTJECT is highly effective in vulnerability injection, almost (99%) code generated by INTJECT are vulnerable. We also address the problem of improving an existing dataset while preserving the intent of the vulnerability in the original dataset. This is useful for improving the effectiveness of learning-based vulnerability injection methods. In addition, we show that our mutation-based data transformation approach is more effective than the baselines, i.e., the original (raw) dataset and obfuscated dataset. Finally, we demonstrate that our approach (INTJECT) generates vulnerable code that is highly semantically similar (in program dependence) to the original vulnerable code.

Our contributions suggest a couple of potential future developments. At the dataset level, we would like to further explore eventual diversification of the program mutations process, in particular, to try other semantic-preserving transformations and operators. At the NLP model level, we would like to explore the benefits of using a transformer approach instead of an RNN. Such powerful models could in principle learn larger vulnerable intents. Using a pre-trained language model (such as CodeBERT) and fine-tuning it for vulnerability could address both large computational needs for training and scarcity of very large vulnerable datasets. Moreover, we plan to conduct a large-scale empirical study to compare the performance and usability of INTJECT to similar vulnerability injection techniques (e.g., Lava and Pitest), using several open-source programs. Finally, we plan to explore the capability of our mutation-based vulnerability injection method (INTJECT) to measure and assess the effectiveness of vulnerability detection methods (such as INFER).

## Acknowledgment

This work was supported by the Luxembourg National Research Fund (FNR) TestFast Project, ref. 12630949, the Fonds National de la Recherche Scientifique (FNRS) EOS VeriLearn project (Grant O05518F-RG03), and the ERDF project IDEES. Gilles Perrouin is an FNRS Research associate.

## References

[1] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 110–121.

[2] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, 1997.

[3] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. L. Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 354–365. [Online]. Available: https://doi.org/10.1145/2931037.2931040

[4] J. Fonseca, M. Vieira, and H. Madeira, "Vulnerability & attack injection for web applications," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 2009, pp. 93–102.

[5] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," *Adv. Comput.*, vol. 112, pp. 275–378, 2019. [Online]. Available: https://doi.org/10.1016/bs.adcom.2018.03.015

[6] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. L. Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Trans. Software Eng.*, vol. 44, no. 4, pp. 308–333, 2018. [Online]. Available: https://doi.org/10.1109/TSE.2017.2684805

[7] T. Loise, X. Devroey, G. Perrouin, M. Papadakis, and P. Heymans, "Towards security-aware mutation testing," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017, pp. 97–102.

[8] L. Liu, Z. Li, Y. Wen, and P. Chen, "Investigating the impact of vulnerability datasets on deep learning-based vulnerability detectors," *PeerJ Computer Science*, vol. 8, p. e975, 2022.

[9] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2021.

[10] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and java test suite," *Computer*, vol. 45, no. 10, pp. 88–90, 2012. [Online]. Available: https://doi.org/10.1109/MC.2012.345

[11] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *Acm sigplan notices*, vol. 39, no. 12, pp. 92–106, 2004.

[12] V. Kashyap, J. Ruchti, L. Kot, E. Turetsky, R. Swords, S. A. Pan, J. Henry, D. Melski, and E. Schulte, "Automated customized bug-benchmark generation," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019, pp. 103–114.

[13] A. Khanfir, A. Koyuncu, M. Papadakis, M. Cordy, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Ibir: Bug report driven fault injection," *ACM Trans. Softw. Eng. Methodol.*, may 2022, just Accepted. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/3542946

[14] W. Bonnaventure, A. Khanfir, A. Bartel, M. Papadakis, and Y. Le Traon, "Confuzzion: A java virtual machine fuzzer for type confusion vulnerabilities," in *IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 12 2021, pp. 586–597.

[15] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods*, K. Havelund, G. Holzmann, and R. Joshi, Eds. Cham: Springer International Publishing, 2015, pp. 3–11.

[16] C. K. Behera and D. L. Bhaskari, "Different obfuscation techniques for code protection," *Procedia Computer Science*, vol. 70, pp. 757–763, 2015, proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050915032780

[17] S. Designs, "Java source code obfuscator," http://www.semdesigns.com/products/obfuscators/JavaObfuscator.html, 5 2022.

[18] OpenNMT, "An open source neural machine translation system," https://opennmt.net/, 5 2022.

[19] P. Black, "Juliet 1.3 test suite: Changes from 1.2," 06 2018.

[20] M. Corporation, "The common weakness enumeration (cwe) initiative," http://cwe.mitre.org/, 5 2022.

[21] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Information and Software Technology*, vol. 68, pp. 18–33, 2015.

[22] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, pp. 1–36, 2017.

[23] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019. [Online]. Available: https://doi.org/10.1145/3340544

[24] A. Dann, B. Hermann, and E. Bodden, "Sootdiff: Bytecode comparison across different java compilers," in *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 14–19. [Online]. Available: https://doi.org/10.1145/3315568.3329966

[25] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[26] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[27] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Learning how to mutate source code from bug-fixes," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 301–312.

[28] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, ser. AAAI'17. AAAI Press, 2017, p. 1345–1351.

[29] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, sep 2019. [Online]. Available: https://doi.org/10.1145/3340544

[30] J. Fonseca and M. Vieira, "Mapping software faults with web security vulnerabilities," in *2008 IEEE international conference on dependable systems and networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 257–266.

[31] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 449–452.

[32] J. Pewny and T. Holz, "Evilcoder: Automated bug insertion," *CoRR*, vol. abs/2007.02326, 2020. [Online]. Available: https://arxiv.org/abs/2007.02326

[33] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[34] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2012.

[35] M. Ojdanic, A. Garg, A. Khanfir, R. Degiovanni, M. Papadakis, and Y. L. Traon, "Syntactic vs. semantic similarity of artificial and real faults in mutation testing studies," *arXiv preprint arXiv:2112.14508*, 2021.

[36] R. Degiovanni and M. Papadakis, "$\mu$bert: Mutation testing using pre-trained language models," 2022. [Online]. Available: https://arxiv.org/abs/2203.03289

[37] A. Khanfir, M. Jimenez, M. Papadakis, and Y. L. Traon, "Codebert-nt: code naturalness via codebert," *arXiv preprint arXiv:2208.06042*, 2022.

[38] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 529–540. [Online]. Available: https://doi.org/10.1145/1315245.1315311

[39] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, Feb. 2013.

[40] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, p. 772–787, Nov. 2011. [Online]. Available: https://doi.org/10.1109/TSE.2010.81

[41] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *J. Syst. Archit.*, vol. 57, no. 3, p. 294–313, Mar. 2011. [Online]. Available: https://doi.org/10.1016/j.sysarc.2010.06.003

[42] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[43] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-2_Li_paper.pdf