Semantic-Preserving Transformations as Mutation Operators: A Study on Their Effectiveness in Defect Detection

Max Hort
Simula Research Laboratory
Oslo, Norway
Email: maxh@simula.no

Linas Vidziunas
Simula Research Laboratory
Oslo, Norway
Email: linasvidz@simula.no

Leon Moonen
Simula Research Laboratory
Oslo, Norway
Email: leon.moonen@computer.org

Abstract—Recent advances in defect detection use language models. Existing works enhanced the training data to improve the models' robustness when applied to semantically identical code (i.e., predictions should be the same). However, the use of semantically identical code has not been considered for improving the tools during their application - a concept closely related to metamorphic testing.

The goal of our study is to determine whether we can use semantic-preserving transformations, analogue to mutation operators, to improve the performance of defect detection tools in the testing stage. We first collect existing publications which implemented semantic-preserving transformations and share their implementation, such that we can reuse them. We empirically study the effectiveness of three different ensemble strategies for enhancing defect detection tools. We apply the collected transformations on the Devign dataset, considering vulnerabilities as a type of defect, and two fine-tuned large language models for defect detection (VulBERTa, PLBART).

We found 28 publications with 94 different transformation. We choose to implement 39 transformations from four of the publications, but a manual check revealed that 23 out 39 transformations change code semantics. Using the 16 remaining, correct transformations and three ensemble strategies, we were not able to increase the accuracy of the defect detection models. Our results show that reusing shared semantic-preserving transformation is difficult, sometimes even causing wrongful changes to the semantics.

Index Terms—defect detection, language model, semantic-preserving transformation, ensemble

I. INTRODUCTION

Among others, defect detection models should be accurate (*good at detecting defective code snippets*) and robust (*making the same prediction for functions that are doing the same thing*). Existing research measured how robust defect detection models are [1] and improved robustness via the training process (e.g., adversarial training) by incorporating code snippets that look different but have the same functionality [1, 2]. One approach to obtain such snippets is the use semantic-preserving transformations (e.g., changing a for loop to a while loop), which do not change the code snippets' functionality. An unexplored aspect is whether "unrobust" models can be improved by using the diverse predictions made on code transformed with semantic-preserving transformations.

This intuition is inspired by metamorphic testing, which instead of ground truth datasets, has access to inputs which should generate the same output [3, 4]. Models are then evaluated by their ability to make the same predictions for such input pairs. Rather than comparing the predictions made on multiple inputs, as done in metamorphic testing, we use them to improve the performance of LLMs for defect detection.

As an example, imagine you have written code which a defect detection model predicts as "correct". You now go ahead and change a variable name. All of a sudden the prediction changes to defective. Surprised, you repeat the same procedure another 5 times but the prediction remains as defective. This raises the question whether to trust the first prediction, as it was performed on the original code snippet or the majority of tests, which were labeled as defective. Following this intuition, we study the following: **What is the impact of semantic-preserving transformations on defect detection tool performance?**

To answer this question, we focus on Large Language Models (LLMs) as defect detection tools. First, we review existing literature that applied semantic-preserving transformations. Then we check whether they share resources to find transformations we can replicate and apply for our experiments. Using LLMs as defect detection models and existing approaches for generating semantic-preserving transformations, we focus on combining the two parts and investigate the ability of semantic-preserving transformations to enhance and improve the defect detection models. For this purpose, we use ensemble techniques, which make predictions under consideration of the original code snippet and mutated variants with semantic-preserving transformations. Figure 1 outlines the current process of using LLMs for defect detection and our novel approach, using semantic-preserving transformations for mutating source code, and ensemble learning. We apply the ensembles to the Devign dataset [5], a dataset for detecting vulnerability defects, which categorises functions as vulnerable and non-vulnerable.

We make the following contributions:
- We collect 28 publications that applied semantic-preserving transformations.

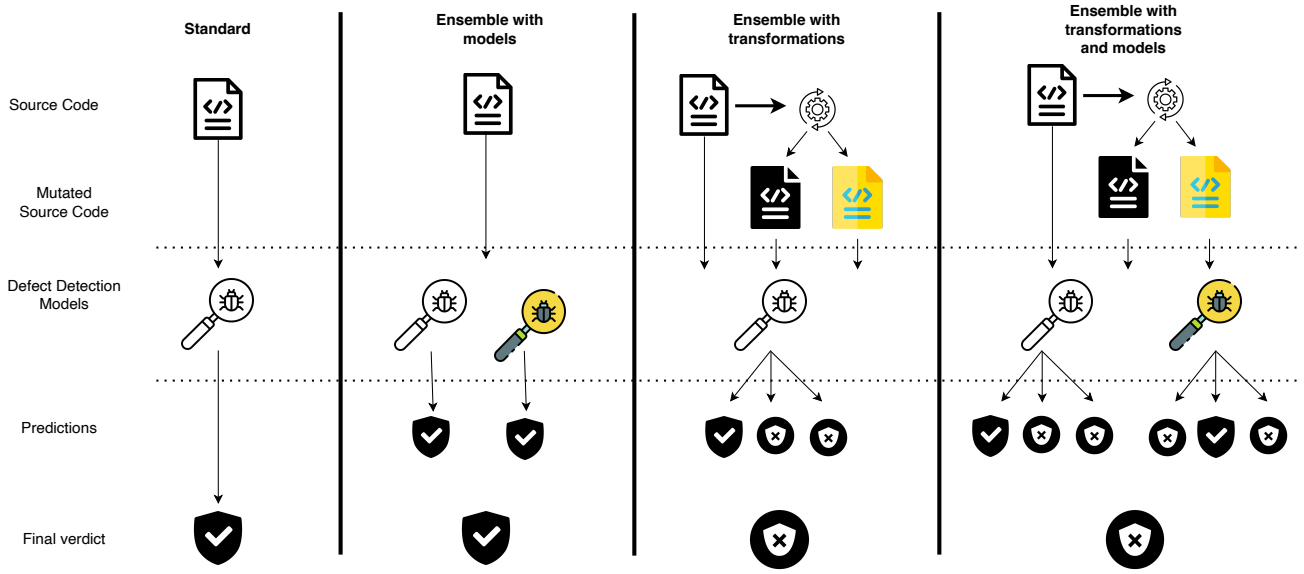1

arXiv:2503.23448v1 [cs.SE] 30 Mar 2025

Fig. 1: Overview of the proposed approach for applying semantic-preserving transformation for testing defect detection models.

- We implement 39 semantic-preserving transformations shared by four publications. For each transformation, we check up to 20 transformed functions manually to verify their correctness.
- We evaluate the usability of semantic-preserving transformations for improving defect detection tools in an empirical study with three ensemble techniques and two LLMs using the transformations.
- We share our results and all transformations applied to the Devign dataset to enable a replication and verification of our results.[1]

Main findings: We have found 28 publications with 94 semantic-preserving transformations. When implementing the transformations of four of these repositories, we encountered difficulties. Critically, 23 out of 39 transformations **did not pass our manual validation, as they changed semantics**. We are left with 16 out of 39 transformations for our empirical study. While our investigated ensemble techniques were not able to improve the accuracy, we provide insights on implementation difficulties (Section V-C).

II. RELATED WORK

A. Data Augmentation for Software Engineering

Data augmentation is used to increase the amount of data available for training or testing of machine learning models. Here we outline approaches for software engineering tasks.

Sampling: The first set of approaches are concerned with the sampling of existing instances in the training dataset. When treating sampling as a data augmentation approach, we focus on the oversampling of data (i.e., the addition of duplicates) rather than downsampling (i.e., removal of samples) [6, 7].

Sampling is in particular useful when dealing with imbalanced datasets, which frequently is the case for vulnerability detection tasks. For instance, datasets might only have 10% of functions labeled as vulnerable [8]. When collecting programs from online sources, such as Github, the proportion of correct samples is much larger than defective programs. Oversampling can balance the two classes [9].

In addition to random sampling, more complex methods have been applied, such as fuzzy-based oversampling [10] or Synthetic Minority Over-Sampling (SMOTE) [11, 12]. SMOTE selects a sample from the under-represented group and one of its closest neighbors (from the same group). The new sample is created by interpolating between the two selected data samples [13]. For such an interpolation to work, one needs to represent programs and code snippets numerically. This has been done with vector embeddings [11, 12] or based on source code metrics [14–16]

Regular oversampling is not able to generate new programs. SMOTE can generate new data but represents programs as vector embeddings rather than source code.

Neural Approaches: Neural approaches use LLMs to learn how to modify existing code snippets and create new snippets from scratch. For example, Richter and Wehrheim [17] used LLMs to learn how to mutate tokens in a code snippet to generate more natural bugs.

Another approach has been followed by Zirak and Hemmati [18], who trained an LLM on a reverse program repair dataset such that they can use the LLM to generate faulty examples (e.g., learn how to create bugs rather than learning how to fix them). For the bug detection and repair task, Allamanis et al. [19] trained two models, one to detect and repair bugs and a second one the learn how to best insert bugs to be used as training data. An adaptation of this approach is introduced by Yasunaga and Jiang [20]. They employed

---

[1] https://figshare.com/s/f9b93d9d549f316b1e5d.

2

the iterative approach break-it-fix-it (BIFI) for data generation. BIFI uses a fixer, which learns to repair buggy code, and a breaker which learns to insert bugs in fixed code. In an iterative process, both these models are used to improve one another (e.g., the fixer learns from the code "broken" by the breaker and vice versa).

While neural approaches are able to generate new source code samples, which look natural, they are usually applied to create datasets for program repair tasks (i.e., insert natural bugs for good code). Thereby, the created programs aim at breaking the code and do not maintain the functionality of the original code snippets.

Rule-based transformations: Previous described approaches for data augmentation generate new, unseen samples but are not able to guarantee their behavior (e.g., neural approaches) or are not applicable to source code (e.g., SMOTE). One approach to address this shortcoming are rule-based transformations. By carefully defining the modifications, one can ensure that the functionality of modified snippets is not altered (e.g., change for loops to while loops). We note that one can also apply rule-based transformations to insert faults, but our focus here is the use of semantic-preserving transformations, transformation which can change the syntax of a code snippet but not the functionality.

Semantic-preserving transformations have been applied to a variety of software engineering tasks, including code clone detection [2] and code authorship disguise [21, 22]. The application ranges from the creation or extension of datasets to robustness evaluation and changes to pre-training procedures. For example, Jing [23] used transformations to increase the complexity of samples in a vulnerability detection dataset. Other works tested the robustness of LLMs to small changes in the input. This means that a model should return the same prediction when faced with a code snippet and a transformed one with the same semantics [24]. Semantic-preserving transformations used to test the robustness can also be used to extend the training dataset and in turn improve a models' robustness [1, 25]. Lastly, semantic-preserving transformation have been used to modify the loss function when training LLMs for software engineering tasks, one example being contrastive pre-training. Contrastive pre-training [2, 26, 27] guides the training of LLMs such that semantically-identical code snippet are embedded close to one another. Thereby, predictions by the model should be less dependent on syntax.

While rule-based semantic-preserving transformations have been applied to test the robustness of models, create or extend training datasets, they have not been applied to enhance and improve the testing itself. In this work, we are the first to use semantic-preserving transformations to extend the testing procedure, similar to metamorphic testing. We apply this to the defect detection task, in particular the detection of vulnerabilities.

B. Ensemble Learning for Defect Detection

Ensemble models combine predictions made by multiple models. This combination is carried out to address limita-

tions of machine learning models and can achieve higher accuracy than a single classifier [28]. In particular, multiple classification models can be used to complement one another, for example in situation where identifiers are able to detect different types of defects [29, 30]. An ensemble of such classifiers could make for more robust defect detection models, for instance when dealing with vulnerabilities.

A survey on ensembles for defect prediction by Matloob et al. [31] found that the most frequent ensemble techniques to improve the performance of defect detection tools were random forests, boosting, and bagging. There was no mention of ensembles of different defect detection tools.

Di Nucci et al. [30] proposed ASCI (Adaptive Selection of ClassIfiers in bug prediction), an approach which instead of combining multiple classifiers into an ensemble, predicts the most suitable classifier to use. This approach was able to outperform a majority voting baseline.

In 2016, Petrić et al. [32] stated that *"Almost all previous work using ensemble techniques in defect prediction rely on the majority voting scheme for combining prediction outputs"*. Instead of applying a majority voting scheme, they used a stacking ensemble. The first layer of a stacking approach contains predictions made by individual classification models. This is used as an input to the second layer, which learns to make the final prediction.

Further approaches for defect detection with ensemble learning used weighing [33, 34], averaging prediction probabilities [35], and stacked learning with different neural network architectures [36]. Barbez et al. [37] used an ensemble of feature extraction tools to generate a comprehensive input for defect detection. Other than predicting whether a function is vulnerable, Ding et al. [38] used ensemble learning to detect vulnerabilities on a statement level.

III. TRANSFORMATIONS

In this section, we present an overview of semantic-preserving transformations that have been applied in existing works. For this, we performed and exploratory literature search in addition to backward snowballing. We found 28 publications that have applied a total of 94 different transformations. Table I and Table II list the 94 transformations. A description of each transformation can be found in our online appendix.[1]

We further divide the transformations in six categories, inspired by Quiring et al. [21] and Liu et al. [22]: API, Formatting, Control Flow, Function, Data and Declaration, Dead/bogus code, Trivial.

API transformations are concerned with API use, mainly for input and output processing. **Formatting** transformations do not change any of the tokens of code snippets, but rather adjust spacing. **Control Flow** transformations describe modifications to the snippets control flow, such as modification of if-statements or the processing order of loop operations. These are often times bi-directional, such as the conversion of for loops to while loops and vice-versa. **Function** transformations are concerned with transformations that address the modification of functions (e.g., function parameters) or the

TABLE I: Semantic-preserving transformations - Part 1.

Types	Transformation	#	Reference
Trivial	Replace variable name	25	[1–4, 19, 21, 22, 24–27, 39–52]
	Replace argument/parameter	8	[1, 2, 4, 42, 44, 46, 50, 53]
	Replace function name	8	[1, 3, 21, 22, 26, 42, 44, 53]
	Object field renaming	3	[41, 46, 50]
	Split/aggregate declarations	3	[22, 49, 54]
	Split/aggregate multi-variable assignment	3	[47, 49, 54]
	API renaming	3	[1, 42, 44]
	Swap operands	3	[19, 22, 47]
	Undo or introduce type alias	3	[21, 22, 49]
	Split/aggregate assignment initialization	2	[49, 54]
	Prefix / suffix operator swap	2	[49, 54]
	Separate/attach elaborated type declaration	1	[22]
	Replace a class	1	[3]
	Property assignment renaming	1	[41]
	Use namespace	1	[49]
	Use macros	1	[49]
	Use alternative tokens	1	[22]
	Use converse-negative expressions	1	[22]
	Use equivalent computations	1	[22]
	Numerical calculation transformation	1	[39]
	Self-increasing/decreasing unfolding	1	[54]
	Ternary expressions	1	[41]
	Array access	1	[41]
	Array indexing/pointer	1	[49]
Data and Declaration	Add neutral element	5	[3, 4, 42–44]
	If enhancement	3	[1, 42, 44]
	Convert int literals into expressions, vice versa	2	[2, 22]
	Convert between bool literals and int literals	2	[21, 22]
	Replace bool literals	2	[46, 50]
	Boolean exchange	2	[24, 45]
	For loop enhancement	2	[42, 44]
	Convert integers into hexadecimal numbers	1	[22]
	Promote data types	1	[21]
	Convert char literals into ASCII values	1	[22]
	Convert string literals to char arrays	2	[21, 22]
	Convert array to vector	1	[21]
	Declaration of loop variables	1	[21]
	Type upconversion	1	[2]
	Use cast expressions	1	[22]
	Use typeid expression	1	[22]
API	Input API transformation	3	[21, 49, 54]
	Output API transformation	3	[21, 49, 54]
	Input interface transformer	2	[21, 49]
	Output interface transformer	2	[21, 49]
	Sync-with-stdio transformer	2	[21, 49]

TABLE II: Semantic-preserving transformations - Part 2.

Types	Transformation	#	Reference
Control flow	Convert for-statement to while-statement	9	[1, 21, 22, 24, 27, 39, 47, 49, 54]
	Convert while-statement to for-statement	9	[1, 21, 22, 24, 27, 39, 47, 49, 54]
	Convert switch-case to if-else	6	[22, 24, 27, 45, 49, 54]
	Reorder statements	5	[24, 26, 27, 45, 49]
	Convert if-else to switch-case	4	[22, 27, 49, 54]
	Split conditions of if-statements	4	[21, 22, 49, 54]
	Swap if-else bodies	3	[19, 22, 47]
	Convert ternary to if	2	[47, 54]
	If-true	2	[3, 4]
	Lambda-identity	2	[3, 4]
	Combine if statement	2	[49, 54]
	Convert if-else to conditional expression	1	[22]
	Convert conditional expression to if-else	1	[22]
	Convert if-else(if) to if-if	1	[39]
	Unroll while loop	1	[50]
	Insert multiple loops	1	[23]
	Wrap Try Catch	1	[50]
	Delegation-method	1	[3]
Function	Add function arguments	5	[1, 3, 22, 42, 44]
	Function creation	3	[21, 49, 53]
	Reorder function arguments	2	[22, 53]
	Add input checking for function parameters	2	[42, 44]
	Merge function arguments	1	[22]
	Convert statements into functions	1	[22]
	Convert binary expressions into functions	1	[22]
	Merge functions	1	[22]
	Hide API calls	1	[22]
	Merge function arguments	1	[22]
Dead/bogus code	Add dead code	12	[2, 23, 26, 27, 42–47, 50, 53]
	Add unused variable	10	[1, 3, 4, 24, 40–44, 53]
	Add temp variables	5	[21–23, 39, 49]
	Add print statement	5	[1, 42, 44, 46, 50]
	Add, remove, move comments	5	[2, 3, 19, 26, 53]
	Return optimal	3	[1, 42, 44]
	Duplication	3	[42–44]
	Add libraries/includes	3	[21, 22, 49]
	Add global declarations	3	[21, 22, 49]
	Remove dead code	3	[2, 21, 22]
	Add return statement	2	[21, 49]
	Add redundant operands	2	[22, 43]
	Add type alias	2	[21, 22]
	Add/remove compound statement	1	[21]
	Add unused object expression	1	[41]
	Add function declarations in classes	1	[22]
	Add void function	1	[53]
Formatting	Add whitespace	2	[3, 53]
	Remove whitespace	1	[3]
	Reformatting	1	[2]
	Beautification	1	[2]
	Compression	1	[2]

creation of new functions (e.g., the conversion of statements to functions or the merging of two functions). **Data and Declaration** addresses transformations such as the declaration of variables (e.g., defining multiple variables in a single statement) and the conversion of different data types (e.g., casting integers to floats). **Dead/bogus Code** describes the addition of useless statements to the code, which do not change the execution, such as duplicating assignments or modifying comments. Unused code can describe a multitude of constructs, such as if, for, if else, switch or while statements [42–44]. **Trivial** transformations include all other, small modifications to the code, such as renaming (functions, variable names, arguments), rewriting of operands (i++ is changed to i+=1), or the splitting/aggregation of statements. The most frequently applied transformation belongs to this category, i.e., the renaming of variable names. Variable renaming has been performed by 25 out of 28 publications, following different strategies, such as the use of synonyms [1], setting the variable name to VAR_i [47], adversarial selection [48] and random generation [43].

A. Publications with Shared Artifacts

In addition to collecting publications with semantic-preserving transformation, we are particularly interested in the ones that share their implementation. A shared implementation allows for a reproduction of the results and application of transformations without incurring large implementation overheads and the risk of errors or different implementation choices. Among the 28 publications that applied semantic-preserving transformations, we found 19 which share code, such that one can apply their transformations. Table III provides details on each of the 19 publications. For each publication, we describe the task to which they have been applied to (e.g., code search, clone detection), the number of implemented transformations (according to Table I and Table II), the respective programming languages as well as a link to their resources.

From Table III, we can observe that the shared transformation have been applied to various tasks, ranging from defects (vulnerability detection, program repair), code translation to

TABLE III: Overview of publications with shared source code for transformations and the respective programming languages (PLs), including the number of implemented transformations (#).

Author, year	Topic	#	PLs	Link
Zhen et al. [49], 2022	Authorship attribution	26	C/C++, Java	https://github.com/RoPGen/RoPGen
Quiring et al. [21], 2019	Authorship attribution	24	C/C++	https://github.com/EQuiw/code-imitator
Dong et al. [55], 2023	Code classification, Defect detection	14	Java, Python	https://github.com/zemingd/Mixup4Code
Applis et al. [3], 2021	Code summarization	12	Java	https://github.com/ciselab/Lampion/tree/main/Transformers
Pour et al. [1], 2021	Method name prediction, code captioning, code search, documentation generation	11	Java	https://github.com/MaryamVP/Guided-Mutation-ICST-2021
Jain et al. [2], 2021	Code clone detection, type inference, Extreme code summarization	10	JavaScript, TypeScript	https://github.com/parasj/contracode
Risse and Boehme [53], 2023	Vulnerability detection	9	C/C++	https://github.com/niklasrisse/LimitsOfML4Vuln
Ramakrishnan et al. [50], 2022	Method name prediction	8	Java, Python	https://github.com/jjhenkel/averloc
Chakraborty et al. [47], 2022	Text to code generation, code translation, bug fixing	8	Java, Python, C, C#, Go, JavaScript, Ruby, PHP	https://github.com/saikat107/NatGen
Rabin et al. [24], 2021	Method name prediction	7	Java	https://github.com/mdrafiqulrabin/tnpa-generalizability/
Bielik et al. [41], 2020	type prediction	7	JavaScript, TypeScript	https://github.com/eth-sri/robust-code
Wei et al. [43], 2021	NL summary, Method name prediction	6	Java	https://zenodo.org/record/4000441#.ZEeX2-xByX0
Srikant et al. [46], 2021	Method name prediction	6	Java, Python	https://github.com/ALFA-group/adversarial-code-generation
Liu et al. [26], 2023	Clone detection, defect detection, code translation, code search	5	Java, Python, Ruby, Go, PHP, JavaScript	https://github.com/shangqing-liu/ContraBERT
Allamanis et al. [19], 2021	Bug detection, program repair	4	Python	https://github.com/microsoft/neurips21-self-supervised-bug-detection-and-repair
Yefet et al. [40], 2020	Code classification, Bug detection	2	Java, C#	https://github.com/tech-srl/adversarial-examples
Zhang et al. [25], 2020	Source code classification	1	C/C++	https://github.com/SEKE-Adversary/MHM
Yang et al. [48], 2022	Vulnerability prediction, Clone detection, Authorship attribution	1	C, Python, Java	https://github.com/soarsmu/attack-pretrain-models-of-code
Yang et al. [51], 2023	Code summarization NL, method name prediction	1	Python	https://figshare.com/articles/dataset/ICSE-23-Replication_7z/20766577/1

code clone detection and authorship attribution. Moreover, there are several publications that applied a single transformation, i.e., variable renaming.

In total, shared transformations have been applied to 9 programming languages. The most frequently used programming language is Java, with 12 out of 19 publications. The next popular programming language is Python (8 publications) followed by C/C++ (6 publications) and JavaScript (4 publications). In 9 out of 19 cases, transformations have been applied to a single programming language, the remaining 10 consider at least two and at most 8.

IV. EMPIRICAL STUDY DESIGN

A. Research Questions

We set out to answer two research questions in our study:

RQ1. What is the impact of semantic-preserving transformation operators on the predictions made by defect detection tools?
In the first RQ, we investigate whether semantic-preserving transformations lead to changes in the predictions of defect detection tools. In particular, we are interested in vulnerability defects in code snippets. This question is important to verify the viability of our approach for improving vulnerability detection (e.g., if the transformed code does not change predictions, the information cannot be used for improvement). Moreover, we investigate how often each transformation can be applied to the functions of the considered dataset (Section IV-B). These insights support our second research question:

RQ2. To what extent can transformation be used to improve defect detection tools?
To answer this question, we augment the defect detection tools, normally applied to a single function, to incorporate predictions performed on the transformations. This resembles an ensemble strategy (e.g., consider the majority prediction of the original and x transformed functions). More details

on the implemented ensemble approaches are provided in Section IV-E.

B. Dataset

We use the Devign [5] vulnerability detection dataset to evaluate semantic preserving transformations, which is part of the ML benchmark collection CodeXGLUE [56]. The Devign dataset [5] includes $27,318$ C/C++ functions from open-source projects, labeled vulnerable or non-vulnerable (54% of the functions are labeled non-vulnerable).[2] Here, the label 1 indicates a vulnerable and the label 0 a non-vulnerable function. The data is split as follows: 80% training, 10% testing, 10% validation.[3] To determine the quality of defect detection tools applied to the Devign dataset, we measure their accuracy (i.e., the proportion of correctly labeled functions), in accordance with CodeXGLUE [56].

C. Models

Table IV provides an overview of all the LLMs applied to the defect detection task on CodeXGLUE [56]. We list LLMs according to their accuracy on the Devign test set and provide details on the type of artifacts shared. For each LLM, source code, pre-trained models and/or fine-tuned variants can be shared. Here, fine-tuned indicates whether a model has been fine-tuned on the Devign training set and shared. We observe that there are only two LLMs for which fine-tuned versions are shared: VulBERTa and PLBART. Therefore, we use VulBERTa-MLP (the better performing variant of both VulBERTa versions) and PLBART to investigate semantic preserving transformations without incurring training costs.

We have run these models on one NVIDIA Tesla V100 SXM3 32 GB GPU.

[2]https://drive.google.com/file/d/1x6hoF7G-tSYxg8AFybggypLZgMGDNHfF
[3] https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection

5

TABLE IV: Vulnerability detection models and their shared resources. Accuracy is given in accordance with CodeXGLUE.

Model	Accuracy	Fine-tuned	Pre-trained	Code	URLs
UniXcoder-nine-MLP [57]	69.29		✓	✓	https://github.com/microsoft/CodeBERT/tree/master/UniXcoder
CoTexT [58]	66.62		✓		https://huggingface.co/razent/cotext-1-ccg
C-BERT [59]	65.45				
A-BERT	65.37				
RefactorBERT	65.08				
VulBERTa-MLP [60]	64.75	✓	✓	✓	https://github.com/ICL-ml4csec/VulBERTa
VulBERTa-CNN [60]	64.42	✓	✓	✓	https://github.com/ICL-ml4csec/VulBERTa
ContraBERT_C [26]	64.17		✓	✓	https://github.com/shangqing-liu/ContraBERT
ContraBERT_G [26]	63.32		✓	✓	https://github.com/shangqing-liu/ContraBERT
PLBART [61]	63.18	✓	✓	✓	https://github.com/wasiahmad/PLBART
code2vec [62, 63]	62.48		✓	✓	https://github.com/dcoimbra/dx2021, https://github.com/tech-srl/code2vec
CodeBERT [64]	62.08		✓	✓	https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection

D. Semantic Preserving Transformations

To determine semantic-preserving transformations to implement from existing works (Table I & II), we first select relevant publications to replicate according to three criteria:

1) Implementation is available.
2) Multiple transformations are implemented.
3) Transformations are applicable to C/C++.

The first stage of filtering leaves us with 19 publications, which can be seen in Table III. Seven of these can be applied to C/C++ code, which is required given the Devign dataset (Section IV-B). Next, we filter publications that implemented only a single transformation. We decided for this filtering to achieve a good trade-off between the effort required to implement shared repositories and the total number of transformations. This leaves us with repositories from four publications: Nat-Gen [47], CodeImitator [21], LimitsOfML4Vuln [53], RoP-Gen [49]. These publications provide a total of 39 transformations, for which we obtained implementations from the shared repositories. Lastly, we perform another filtering stage to determine which of these 39 transformations to keep:

4) A manual check confirms that the transformations are indeed semantically-preserving.

For this purpose, we apply the transformations to functions of the Devign dataset and manually checked the correctness of up to 20 of them (i.e., we checked at least 20 transformed functions to confirm correctness but might terminate earlier when we have encountered incorrect transformations). We perform this check for each the the 39 transformations. If any of those **did change the semantics of the function**, we excluded the transformation for further experiments, which was the case for 23 of them. We are left with 16 transformations without encountered errors. These constitute all transformation that we could implement from the four existing publications without encountering errors (i.e., the other 23 transformations changed semantics contrary to their claim). We provide an overview of the investigated transformed functions and our verdict, whether they are semantic-preserving, in our online appendix.[1]

E. Ensemble Strategies

We consider three types of ensembles (see Figure 1):

1) Ensemble with a single model and multiple functions;

2) Ensemble with multiple models and a single function;
3) Ensemble with multiple models and multiple functions.

We follow popular ensemble approaches, such as majority voting and weighting [28, 65]. **Majority Voting** makes predictions based on the majority prediction from a collection of prediction (e.g., if 7 out of 10 predictions are positive, the final verdict is positive). We consider two variants of majority voting, depending on the strategy to break ties: selecting the label 0 or the label 1.

The second strategy we employ is **averaging**. Rather than considering predictions as binary values (either 0 or 1), averaging uses the predicted probabilities (ranging from 0 to 1), and averages them among all available predictions. A potential advantage of averaging is that it takes "certainty" in account, e.g., a value of 0.52 receives the prediction 1 while being close to the decision boundary (0.5) is treated identical to 0.98.

Lastly, we employ a **weighting** strategy. Weighting can be compared to stacking, which builds a prediction model on top of the available predictions. While majority voting and averaging give an equal weight to all available predictions, weighting assigns specific weights to each:

$$pred = w_{17} * predVulberta + w_{18} * predPlbart + \sum_{n=1}^{16} w_n * predT_n \quad (1)$$

Here, $predT_n$ shows the predictions made on a transformed function, using transformation Tn. $predVulberta$ and $predPlbart$ are the predictions performed on the original function by the two LLMs. The final prediction ($pred$) requires a total of 18 weights ($w_1, ..., w_{18}$): original prediction from VulBERTa, prediction from PLBART, 16 transformations.[4]

We follow two strategies to handle weighting: based on labels, based on probabilities. When treating the predictions based on **labels**, we convert the label 0 to -1 while a label of 1 remains unchanged. This allows us to sum up predictions and better aggregate the predictions obtained on transformed functions. For instance, if T_1 can be applied to a function four times, for which converted labels are [-1,-1,-1,1], $predT_1$ is assigned a score equal to the sum of predicted labels:

---

[4]In our implementation, w_1 and w_2 are used for the original functions and the later 16 for the transformations.
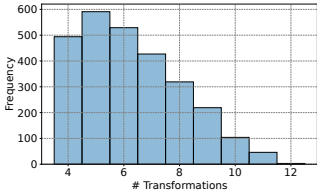
Fig. 2: RQ1: Number of transformations per function in the Devign test set.

$predT_1 = \sum[-1, -1, -1, 1] = -2$. Without converting labels, we would receive $predT_n = \sum[0, 0, 0, 1] = 1$. Similarly, we adjust predictions based on **probabilities** in a range of -0.5 to 0.5. We do this by subtracting 0.5 from the probability of predicting the label 1. For example, if the probabilities to predict each of the two labels is $[0.2, 0.8]$, we treat it as 0.3 $(0.8 - 0.5)$.

To find the weights, we treat the Equation 1 as an optimization problem to find weights which maximize accuracy on the validation set. We use optimizers from SciPy [66] to obtain the weights.[5]

F. Threats to Validity

Here we address threats with regards to our implementation and analysis (threats to internal validity) and the generalizability of results (threats to external validity).

To reduce threats to internal validity, we use existing repositories and implementations for our experiments. In particular, every semantic-preserving transformation we applied is based on the implementation shared by published work. To verify whether the transformations indeed preserve semantics, we carried out a manual check of up to 20 transformed functions. While we were able to exclude 23 incorrect transformations in this way, there might still be undiscovered faults which more extensive checks might be able to expose. Moreover, the two investigated LLM have already been fine-tuned, removing the risk of any mistakes in the training procedure. Given the benchmarking provided by CodeXGLUE and the fact that the Devign dataset is balanced, we opted for using accuracy to determine the performance of LLMs. However, we note that other metrics, such as F-score, could be considered as well. Lastly, we share our results online to allow for replicability.

To reduce threats to external validity, we applied a wide range of semantic-preserving transformations, three ensemble strategies and two pre-trained LLMs. The performance is evaluated on the Devign dataset, a defect detection datasets with real-world bugs. An extension of experimental artifacts could further improve the generalizability of results.

V. RESULTS & DISCUSSION

A. RQ1: Semantic-Preserving Transformations

In this research question, we address how the 16 semantic preserving transformations impact the predictions made by two

---

[5]We used the following eight optimizers: Nelder-Mead,Powell,CG,BFGS,L-BFGS-B,TNC,COBYLA,SLSQP

LLMs (VulBERTa, PLBART). For this purpose, we apply each transformation to every function in the Devign test set, if they are applicable.

First, we consider how frequently each transformation can be applied. Figure 2 illustrates the number of transformations applied to each of the 2732 functions in the Devign test set. We can observe that, each function can be transformed 6.3 times on average, with values ranging from 4 to 12. Only three functions received 12 transformations, which means that we obtain 12 transformed variants on which a single transformation has been applied to.

Figure 3 (leftmost chart) shows how often each of the transformation can be applied. We were able to apply four out 16 transformations to each of the functions in the Devign test set. These modify whitespaces, comments and dead code. Transformation to the initialization of variables, as provided by RopGen, can be applied to 60% of the functions. The remaining transformations can only be applied rarely, with some being applied only three times. Moreover, Figure 3 presents the predictions made on the transformed functions and whether they change the prediction of the original function. In particular, we illustrate whether a predicted label remains at 1 ("1->1"), changes from a 1 to a 0 ("1->0"), changes from a 0 to a 1 ("0->1") or remains at 0 ("0->0"). 1 indicates that a function was labeled as vulnerable, while a label of 0 indicates a non-vulnerable function. Here, the ratios of the four groups varies between the respective transformations with 51% of labels not changing after applying transformations (either "0->0" or "1->1"). The behavior differs between VulBERTa and PLBART, as can be seen in the $insert\_unexecuted\_code$ (LimitsOfML4Vuln). The majority of functions labeled as non-vulnerable by VulBERTa remain unchanged, while PLBART is more likely to predict them as vulnerable. We continue to investigate the ability of transformations to correct defect detection tools in RQ2.

Answer to RQ1: Semantic-preserving transformation change predictions made by the two LLMs in 49% of the cases. These can either change vulnerable to non-vulnerable predictions (25%) or vice-versa (24%).

B. RQ2: Defect Detection Ensembles

In the second research question, we investigate the use of semantic-preserving transformations to enhance the testing of defect detection tools. For this purpose, we employ three different ensemble strategies (Section IV-E) based on combinations of transformations and LLMs. Table V compares the accuracy of the original defect detection tools (VulBERTa, PLBART) with the accuracy of the ensemble approaches. Unfortunately, we observe that our ensemble strategies have not been able to increase the accuracy over the original VulBERTa model while performing better than PLBART.

Among the two strategies for ties, breaking ties at 0 always performs better than breaking ties at 1, meaning that the a predicted label of 0 (non-vulnerable) is preferred. This can be explained by the fact that the majority class is 0, and thereby
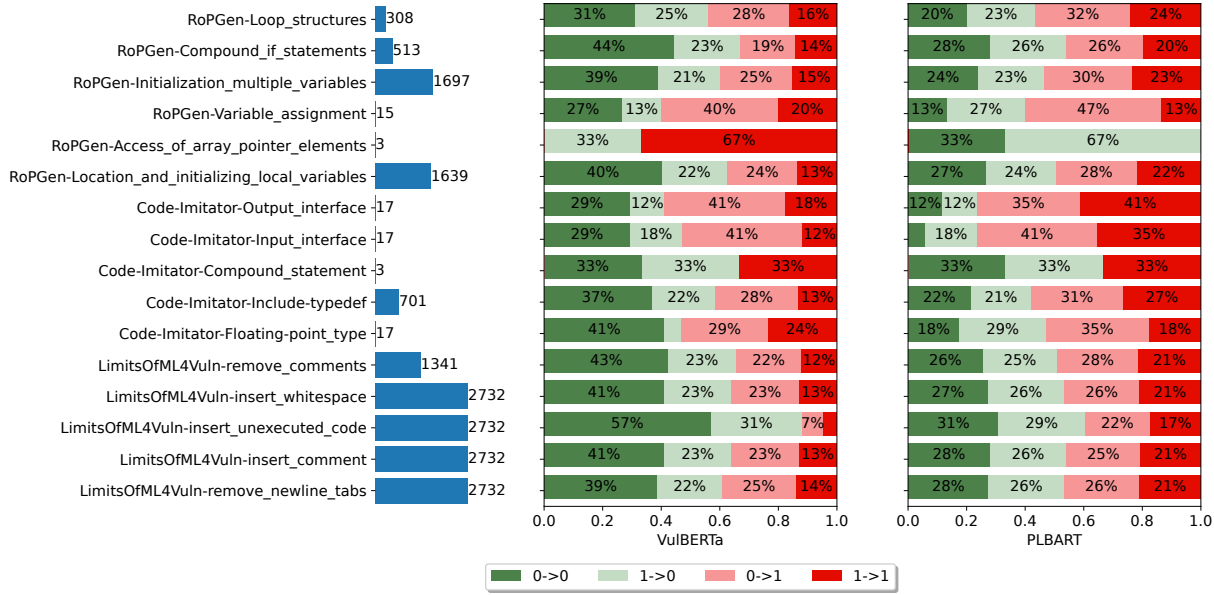
Fig. 3: RQ1: Illustration of the frequency each transformation can be applied (left) and resulting impact on the predicted labels for VulBERTa and PLBART (center and right). For example, "0->1" means that prediction for the original function was 0 and is 1 for the transformation. Here, 0 means the function is non-vulnerable and 1 means vulnerable.

more functions should be labeled as non-vulnerable to achieve a high accuracy. Weighted approaches seem beneficial when dealing with a higher number inputs (i.e., the 16 transformations). However, the highest accuracy of ensemble approaches is achieved by averaging the two LLM predictions, without considering transformations. A reason for this could be that results are difficult to generalize.

Generally speaking, averaging prediction probabilities performs better than simply relying on labels for voting. The advantage of averaging is that the certainty of predictions is taken into account.

Answer to RQ2: Our investigated ensemble strategies were not able to increase the accuracy of defect detection tools.

C. Replication Difficulty

We have found 94 transformations from 28 publications (Section III). Among these, 9 publications did not share their code, which complicates a replication. The remaining 19 studies consider different programming languages, which limits their applicability. In the end, we considered four publications in detail, as they are applicable for C/C++ code and implemented several transformations. 39 of the transformations provided by the four publications were applicable to the Devign dataset (e.g., changes to functions). We were able to re-implement and confirm the semantic-preserving behavior for 16 of these (i.e., by manually checking up to 20 results for each transformation), none of which were part of NatGen [47].

We found that transformations can be task-dependent and therefore not always easy to transfer. For example, Quiring et al. [21] applied transformations for authorship disguise. At

TABLE V: RQ2: Performance of ensemble strategies.

		VulBERTa	PLBART
	Original	64.71	61.79
	CodeXGLUE	64.75	63.18
Data ensemble	Majority - Ties 0	51.43	48.13
	Majority - Ties 1	51.39	47.91
	Average	52.49	50.84
	Weighted - Labels	52.34	54.28
	Weighted - Probability	52.12	59.19
Model ensemble	Majority - Ties 0	58.86	
	Majority - Ties 1	54.54	
	Average	62.52	
	Weighted - Labels	61.79	
	Weighted - Probability	61.75	
Data and model	Majority - Ties 0	52.12	
	Majority - Ties 1	51.79	
	Average	52.12	
	Weighted - Labels	60.03	
	Weighted - Probability	60.61	

times, this requires the presence of a source file (file to be transformed) and target file (style information). Given that we only have a single function at hand, we were not able to use all available transformations.

We have also observed some edge cases, which do not seem to have been considered in the original publications. This can be explained by the different datasets being used. For instance, functions in the Devign dataset can contain assembly code which the variable renaming function of RoPGen did not consider. Another example can be seen in the following code snippet which outlines an erroneous transformation of moving variable definitions inside control statements:

```
1   // Before transformation
2   unsigned i;
3   for (i = 0; i < 10; i++)
4       foo();
5   for (i = 0; i < 10; i++)
6       bar();
7
8   // After transformation
9   for (unsigned i = 0; i < 10; i++)
10      foo();
11  for (i = 0; i < 10; i++)
12      bar();
```

As can be seen from this example, the second for loop, after transformation, is referring to the variable i, however it is only defined in the loop and therefore causes errors due to accessing an undefined variable.

Lastly, the scope of transformations can be different. For example, some transformations are applied on a file-level, while we are focused on functions. Transformations that require files, such as the addition of imports, the use of main functions, or the presence of multiple functions in general, could not be implemented. This shows that one needs to be careful when transferring semantic-preserving transformations from one application to another.

> **Replicability:** Existing semantic-preserving transformations are difficult to replicate due to missing implementations, different programming languages and application scenarios as well as potential incorrect behavior when dealing with edge cases.

VI. CONCLUSIONS

We investigated the use of semantic-preserving transformations for mutating source code to enhance the testing of defect detection tools. We first searched and found 28 publications which implemented such transformations and picked four to implement for our experiments. Overall, it appears challenging to reuse existing methods, due to differences in application scenarios, programming languages or errors due to edge cases. In the end, we were able to successfully use 16 out of 39 available transformations and used them in ensemble approaches for defect detection. While we provide insights on the usefulness of transformations and replication difficulty, we were not able to improve the accuracy of the considered defection detection tools with mutations and ensembles.

Future work efforts can strive to extend the empirical evaluation of transformations for defect detection. This includes additional LLMs, datasets and transformations (e.g., error-inducing transformations). Moreover, one could apply more than a single transformation for each of the functions [1, 21].

The 94 transformations we found stem from 28 publications, some of which do not provide implementations. Therefore, we believe a consolidated framework of semantic-preserving transformation, for different modalities (function or file-level) and programming languages, could benefit the community. One important consideration of such a framework is the provision of proofs to guarantee the correctness of transformations.

REFERENCES

[1] M. V. Pour et al. *A Search-Based Testing Framework for Deep Neural Networks of Source Code Embedding*. 2021. DOI: 10.48550/arXiv.2101.07910.

[2] P. Jain et al. "Contrastive Code Representation Learning." In: *Conf. Empirical Methods in Natural Lang. Processing*. ACL, 2021, pp. 5954–5971. DOI: 10.18653/v1/2021.emnlp-main.482.

[3] L. Applis et al. "Assessing Robustness of ML-Based Program Analysis Tools Using Metamorphic Program Transformations." In: *IEEE/ACM Int'l Conf. Autom. Softw. Eng.* IEEE, 2021, pp. 1377–1381. DOI: 10.1109/ase51524.2021.9678706.

[4] L. Applis et al. "Searching for Quality: Genetic Algorithms and Metamorphic Testing for Software Engineering ML." In: *Genetic & Evolutionary Computation Conf.* ACM, 2023, pp. 1490–1498. DOI: 10.1145/3583131.3590379.

[5] Y. Zhou et al. "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks." In: *Int'l Conf. Neural Information Processing Sys.* Curran Associates, Inc., 2019, p. 11.

[6] M. Zagane et al. "Deep Learning for Software Vulnerabilities Detection Using Code Metrics." In: *IEEE Access* 8 (2020), pp. 74562–74570. DOI: 10.1109/access.2020.2988557.

[7] A. Aumpansub and Z. Huang. "Detecting Software Vulnerabilities Using Neural Networks." In: *Int'l Conf. Machine Learning & Computing*. ACM, 2021, pp. 166–171. DOI: 10.1145/3457682.3457707.

[8] R. Ferenc et al. "Challenging Machine Learning Algorithms in Predicting Vulnerable JavaScript Functions." In: *IEEE/ACM 7th Int'l Ws. Realizing Artificial Intelligence Synergies in Softw. Eng.* IEEE, 2019, pp. 8–14. DOI: 10.1109/raise.2019.00010.

[9] G. Partenza et al. "Automatic Identification of Vulnerable Code: Investigations with an AST-Based Neural Network." In: *IEEE 45th Annual Comp.s, Softw., & Applic. Conf.* IEEE, 2021, pp. 1475–1482. DOI: 10.1109/compsac51774.2021.00219.

[10] S. Liu et al. "DeepBalance: Deep-Learning and Fuzzy Oversampling for Vulnerability Detection." In: *IEEE Trans. Fuzzy Sys.* 28.7 (2020), pp. 1329–1343. DOI: 10.1109/tfuzz.2019.2958558.

[11] S. Jeon and H. K. Kim. "AutoVAS: An Automated Vulnerability Analysis System with a Deep Learning Approach." In: *Computers & Security* 106 (2021), p. 102308. DOI: 10.1016/j.cose.2021.102308.

[12] S. Chakraborty et al. *Deep Learning Based Vulnerability Detection: Are We There Yet?* 2020. DOI: 10.48550/arXiv.2009.07235.

[13] N. V. Chawla et al. "SMOTE: Synthetic Minority Over-sampling Technique." In: *J. Artificial Intelligence Research* 16 (2002), pp. 321–357. DOI: 10.1613/jair.953.

[14] L. Kumar and A. Sureka. "Application of LSSVM and SMOTE on Seven Open Source Projects for Predicting Refactoring at Class Level." In: *Proceedings - Asia-Pacific Softw. Eng. Conf., APSEC*. Vol. 2017-December. 2018, pp. 90–99. DOI: 10.1109/apsec.2017.15.

[15] L. Kumar et al. "Method Level Refactoring Prediction on Five Open Source Java Projects Using Machine Learning Techniques." In: *Innovations on Softw. Eng. Conf.* ISEC'19. ACM, 2019. DOI: 10.1145/3299771.3299777.

[16] F. Pecorelli et al. "On the Role of Data Balancing for Machine Learning-Based Code Smell Detection." In: *3rd ACM SIGSOFT Int'l Ws. Machine Learning Techniques for Softw. Quality Evaluation*. MaLTeSQuE 2019. ACM, 2019, pp. 19–24. DOI: 10.1145/3340482.3342744.

[17] C. Richter and H. Wehrheim. "Learning Realistic Mutations: Bug Creation for Neural Bug Detectors." In: *IEEE Conf. Softw. Testing, Verif. & Validation*. IEEE, 2022, pp. 162–173. DOI: 10.1109/icst53961.2022.00027.

[18] A. Zirak and H. Hemmati. *Improving Automated Program Repair with Domain Adaptation*. 2022. DOI: 10.48550/arXiv.2212.11414.

[19] M. Allamanis et al. "Self-Supervised Bug Detection and Repair." In: *Advances in Neural Information Processing Sys*. Vol. 34. Curran Associates, Inc., 2021, pp. 27865–27876.

[20] M. Yasunaga and P. Liang. "Break-It-Fix-It: Unsupervised Learning for Program Repair." In: *Int'l Conf. Machine Learning*. PMLR, 2021, p. 12.

[21] E. Quiring et al. *Misleading Authorship Attribution of Source Code Using Adversarial Learning*. 2019. DOI: 10.48550/arXiv.1905.12386.

[22] Q. Liu et al. "A Practical Black-Box Attack on Source Code Authorship Identification Classifiers." In: *IEEE Trans. Information Forensics and Security* 16 (2021), pp. 3620–3633. DOI: 10.1109/tifs.2021.3080507.

[23] D. Jing. "Improvement of Vulnerable Code Dataset Based on Program Equivalence Transformation." In: *J. Physics: Conference Series* 2363.1 (2022), p. 012010. DOI: 10.1088/1742-6596/2363/1/012010.

[24] M. R. I. Rabin et al. "On the Generalizability of Neural Program Models with Respect to Semantic-Preserving Program Transformations." In: *Information and Softw. Technology* 135 (2021), p. 106552. DOI: 10.1016/j.infsof.2021.106552.

[25] H. Zhang et al. "Generating Adversarial Examples for Holding Robustness of Source Code Processing Models." In: *AAAI Conf. Artificial Intelligence*. Vol. 34. 2020, pp. 1169–1176. DOI: 10.1609/aaai.v34i01.5459.

[26] S. Liu et al. "ContraBERT: Enhancing Code Pre-Trained Models via Contrastive Learning." In: *Int'l Conf. Softw. Eng*. ICSE '23. regexpIEEE, 2023, pp. 2476–2487. DOI: 10.1109/icse48619.2023.00207.

[27] N. D. Q. Bui et al. "Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations." In: *Int'l ACM SIGIR Conf. Research & Development in Information Retrieval*. 2021, pp. 511–521. DOI: 10.1145/3404835.3462840.

[28] I. D. Mienye and Y. Sun. "A Survey of Ensemble Learning: Concepts, Algorithms, Applications, and Prospects." In: *IEEE Access* 10 (2022), pp. 99129–99149. DOI: 10.1109/access.2022.3207287.

[29] A. Panichella et al. *Cross-Project Defect Prediction Models: L'union Fait La Force*. 2014. DOI: 10.1109/CSMR-WCRE.2014.6747166.

[30] D. Di Nucci et al. "Dynamic Selection of Classifiers in Bug Prediction: An Adaptive Method." In: *IEEE Trans. Emerging Topics in Computational Intelligence* 1.3 (2017), pp. 202–212. DOI: 10.1109/tetci.2017.2699224.

[31] F. Matloob et al. "Software Defect Prediction Using Ensemble Learning: A Systematic Literature Review." In: *IEEE Access* 9 (2021), pp. 98754–98771. DOI: 10.1109/access.2021.3095559.

[32] J. Petrić et al. "Building an Ensemble for Software Defect Prediction Based on Diversity Selection." In: *ACM/IEEE Int'l Symp. Empirical Softw. Eng. & Measurement*. ACM, 2016, pp. 1–10. DOI: 10.1145/2961111.2962610.

[33] Z. Li et al. "Heterogeneous Defect Prediction Through Multiple Kernel Learning and Ensemble Learning." In: *IEEE Int'l Conf. Softw. Maintenance & Evolution*. IEEE, 2017, pp. 91–102. DOI: 10.1109/icsme.2017.19.

[34] Y. Dai et al. "SMASH: A Malware Detection Method Based on Multi-Feature Ensemble Learning." In: *IEEE Access* 7 (2019), pp. 112588–112597. DOI: 10.1109/access.2019.2934012.

[35] I. Laradji et al. "Software Defect Prediction Using Ensemble Learning on Selected Features." In: *Information and Softw. Technology* 58 (2014). DOI: 10.1016/j.infsof.2014.07.005.

[36] L. Wang et al. "PreNNsem: A Heterogeneous Ensemble Learning Framework for Vulnerability Detection in Software." In: *Applied Sciences* 10.22 (2020), p. 7954. DOI: 10.3390/app10227954.

[37] A. Barbez et al. *A Machine-learning Based Ensemble Method For Anti-patterns Detection*. 2019. DOI: 10.48550/arXiv.1903.01899.

[38] Y. Ding et al. *VELVET: A noVel Ensemble Learning Approach to Automatically Locate VulnErable sTatements*. 2022. DOI: 10.48550/arXiv.2112.10893.

[39] Z. Tian et al. *Adversarial Attacks on Neural Models of Code via Code Difference Reduction*. 2023. DOI: 10.48550/arXiv.2301.02412.

[40] N. Yefet et al. "Adversarial Examples for Models of Code." In: *Proceedings of the ACM on Progr. Languages* (2020), pp. 1–30. DOI: 10.1145/3428230.

[41] P. Bielik and M. Vechev. "Adversarial Robustness for Code." In: *Int'l Conf. Machine Learning*. PMLR, 2020, pp. 896–907.

[42] Z. Dong et al. *Boosting Source Code Learning with Data Augmentation: An Empirical Study*. 2023. DOI: 10.48550/arXiv.2303.06808.

[43] M. Wei et al. *CoCoFuzzing: Testing Neural Code Models with Coverage-Guided Fuzzing*. 2021. DOI: 10.48550/arXiv.2106.09242.

[44] Z. Dong et al. *Enhancing Code Classification by Mixup-Based Data Augmentation*. 2022. DOI: 10.48550/arXiv.2210.03003.

[45] E. Shi et al. *Enhancing Semantic Code Search with Multimodal Contrastive Learning and Soft Data Augmentation*. 2022. DOI: 10.48550/arXiv.2204.03293.

[46] S. Srikant et al. *Generating Adversarial Computer Programs Using Optimized Obfuscations*. 2021. DOI: 10.48550/arXiv.2103.11882.

[47] S. Chakraborty et al. "NatGen: Generative Pre-Training by "Naturalizing" Source Code." In: *ACM Joint Eur. Softw. Eng. Conf. & Symp. Found. Softw. Eng*. ESEC/FSE 2022. ACM, 2022, pp. 18–30. DOI: 10.1145/3540250.3549162.

[48] Z. Yang et al. "Natural Attack for Pre-trained Models of Code." In: *Int'l Conf. Softw. Eng*. 2022, pp. 1482–1493. DOI: 10.1145/3510003.3510146.

[49] Z. Li et al. "RoPGen: Towards Robust Code Authorship Attribution via Automatic Coding Style Transformation." In: *Int'l Conf. Softw. Eng*. ACM, 2022, pp. 1906–1918. DOI: 10.1145/3510003.3510181.

[50] G. Ramakrishnan et al. "Semantic Robustness of Models of Source Code." In: *IEEE Int'l Conf. Softw. Analysis, Evolution & Reeng*. 2022, pp. 526–537. DOI: 10.1109/saner53432.2022.00070.

[51] Z. Yang et al. *Stealthy Backdoor Attack for Code Models*. 2023. DOI: 10.48550/arXiv.2301.02496.

[52] J. M. Springer et al. *STRATA: Simple, Gradient-Free Attacks for Models of Code*. 2021. DOI: 10.48550/arXiv.2009.13562.

[53] N. Risse and M. Böhme. *Limits of Machine Learning for Automatic Vulnerability Detection*. 2023. DOI: 10.48550/arXiv.2306.17193.

[54] J. Tian et al. "Generating Adversarial Examples of Source Code Classification Models via Q-Learning-Based Markov Decision Process." In: *IEEE 21st Int'l Conf. Softw. Quality, Reliability & Security*. IEEE, 2021, pp. 807–818. DOI: 10.1109/qrs54544.2021.00090.

[55] Z. Dong et al. *MIXCODE: Enhancing Code Classification by Mixup-Based Data Augmentation*. 2023.

[56] S. Lu et al. "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation." In: *Neural Information Processing Sys. Track on Datasets & Benchmarks*. 2021, pp. 1–16.

[57] D. Guo et al. *UniXcoder: Unified Cross-Modal Pre-training for Code Representation*. 2022. DOI: 10.48550/arXiv.2203.03850.

[58] L. Phan et al. "CoTexT: Multi-task Learning with Code-Text Transformer." In: *Ws. Natural Lang. Processing for Prog*. ACL, 2021, pp. 40–47. DOI: 10.18653/v1/2021.nlp4prog-1.5.

[59] L. Buratti et al. *Exploring Software Naturalness through Neural Language Models*. 2020. DOI: 10.48550/arXiv.2006.12641.

[60] H. Hanif and S. Maffeis. *VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection*. 2022. DOI: 10.48550/arXiv.2205.12424.

[61] W. Ahmad et al. "Unified Pre-training for Program Understanding and Generation." In: *Conf. of the North American Chapter of the Association for Computational Linguistics: Human Lang. Technologies*. ACL, 2021, pp. 2655–2668. DOI: 10.18653/v1/2021.naacl-main.211.

[62] U. Alon et al. "Code2vec: Learning Distributed Representations of Code." In: *Princ. Prog. Lang*. ACM, 2019, pp. 1–29. DOI: 10.1145/3290353.

[63] D. Coimbra et al. *On Using Distributed Representations of Source Code for the Detection of C Security Vulnerabilities*. 2021. DOI: 10.48550/arXiv.2106.01367.

[64] Z. Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages." In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. 2020, pp. 1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139.

[65] Y. Yang et al. "A Survey on Ensemble Learning under the Era of Deep Learning." In: *Artificial Intelligence Review* 56.6 (2023), pp. 5545–5589. DOI: 10.1007/s10462-022-10283-5.

[66] P. Virtanen et al. "SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python." In: *Nature Methods* 17.3 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.