ABSTRACT

Automatic program understanding and generation techniques could significantly advance the productivity of programmers and have been widely studied by academia and industry. Recently, the advent of pre-trained paradigm enlightens researchers to develop general-purpose pre-trained models which can be applied for a broad range of program understanding and generation tasks. Such pre-trained models, derived by self-supervised objectives on large unlabelled corpora, can be fine-tuned in downstream tasks (such as code search and code generation) with minimal adaptations. Although these pre-trained models claim superiority over the prior techniques, they seldom follow equivalent evaluation protocols, e.g., they are hardly evaluated on the identical benchmarks, tasks, or settings. Consequently, there is a pressing need for a comprehensive study of the pre-trained models on their effectiveness, versatility as well as the limitations to provide implications and guidance for the future development in this area. To this end, we first perform an extensive study of eight open-access pre-trained models over a large benchmark on seven representative code tasks to assess their reproducibility. We further compare the pre-trained models and domain-specific state-of-the-art techniques for validating pre-trained effectiveness. At last, we investigate the robustness of the pre-trained models by inspecting their performance variations under adversarial attacks. Through the study, we find that while we can in general replicate the original performance of the pre-trained models on their evaluated tasks and adopted benchmarks, subtle performance fluctuations can refute the findings in their original papers. Moreover, none of the existing pre-trained models can dominate over all other models. We also find that the pre-trained models can significantly outperform non-pre-trained state-of-the-art techniques in program understanding tasks. Furthermore, we perform the first study for natural language-programming language pre-trained model robustness via adversarial attacks and find that a simple random attack approach can easily fool the state-of-the-art pre-trained models and thus incur security issues. At last, we also provide multiple practical guidelines for advancing future research on pre-trained models for program understanding and generation.

CCS CONCEPTS
• Software and its engineering → Reusability.

KEYWORDS
Code Representation, Deep Learning, Pre-Trained Language Models, Adversarial Attack

ACM Reference Format:

1 INTRODUCTION

The research tasks of program understanding and generation, e.g., code summarization, code generation, code search, and defect prediction, have been increasingly studied for decades. Moreover, the advent of deep learning and machine learning techniques has strongly advanced the progress of such research domains [9, 25, 26, 50, 84]. Recently, a group of researchers have proposed natural language (NL)-programming language (PL) pre-trained models to provide general-purpose representations of the semantic connections between natural languages and program languages to support various program understanding and generation tasks once and
for all [3, 16, 18, 21, 27]. In contrast to conventional deep learning techniques which require task-specific feature extraction or model selection, such NL-PL pre-trained models are usually derived from self-supervised language modeling tasks on large unlabelled program language corpora with simple modification and low-cost fine-tuning. Specifically, they tend to adopt the popular transformer [69] architectures, e.g., BERT [15] and GPT [55], for pre-training to facilitate the performance on program understanding and generation tasks. For instance, CodeBERT [18] is pre-trained on millions of program functions and natural language comments by two self-supervised tasks—Masked Language Modeling (i.e., randomly masking input tokens) and Replaced Token Detection [13] (i.e., randomly replacing input tokens) to predict the masked/replaced tokens for input tokens. After pre-training, the CodeBERT model is fine-tuned on different tasks. Moreover, the recently proposed CodeX[12], a GPT-based model pre-trained on massive code samples from Github, automatically fixed 28.8% of programming problems in the HumanEval[12] dataset on the automatic programming task.

Despite the effectiveness and versatility of the NL-PL pre-trained models shown in their original papers, their performance evaluations can nevertheless be susceptible to bias. Specifically, while there are many potential tasks which the NL-PL pre-trained models can be applied for [24, 54, 65, 66, 83], they hardly are applied for fully identical tasks for comprehensive performance comparison and analytics. Moreover, even for certain identical tasks they are evaluated upon, they sometimes adopt diverse benchmarks such that their performance can hardly be precisely compared either. Furthermore, many studies have addressed that general-purpose language models for NL are essentially defective. For instance, their efficacy can be hard to reproduce [6]. In addition, they are quite sensitive to noises and attacks under a set of behavioral tests [59]. Such facts altogether can indicate a strong need of an extensive study for the existing NL-PL pre-trained models.

In this paper, to our best knowledge, we conduct the first comprehensive study on the existing NL-PL pre-trained models to enhance the understanding of their strengths and limitations. Specifically, our objectives of the study include (1) validating the performance of different NL-PL pre-trained models, (2) comparing such models with the previous domain-specific state-of-the-art (SOTA) models, and (3) investigating the robustness of pre-trained models. Note that although the existing work, e.g., CodeXGLUE [43], also presents experimental studies on NL-PL pre-trained models, they hardly evaluate all studied models simultaneously on universal benchmarks and fail to provide insightful analyses on them.

To this end, we first determine to extensively study the effectiveness of the pre-trained models with expanded tasks and datasets which have not been explored by any previous work. In particular, we adopt eight mainstream pre-trained models, i.e., CodeBERT [18], CodeGPT [43], CodeT5 [71], CodeTrans [16], ContraCode [27], CoTeXT [53] GraphCodeBERT [21], and PLBART [3] as our studied subjects mainly because they are publicly available SOTA models designed for a broad spectrum of the program understanding and generation tasks. We then evaluate all the studied pre-trained models on top of benchmark CodeXGLUE [43] because its inclusive datasets have been widely adopted by many of the studied models and it also presents deterministic training/testing data split for strengthening the fairness of their performance comparison. Meanwhile, we evaluate the effectiveness of the studied models against the non-pre-trained SOTA techniques of individual program understanding and generation tasks on top of an extended benchmark. Next, we further inspect the robustness of the studied models via adversarial attacks with multiple existing approaches and a simple random attack approach proposed in our paper.

Our study exposes multiple important and interesting insights for designing and developing future NL-PL pre-trained models. Specifically, we find that no pre-trained models can dominate the other models on all studied downstream tasks and their validity can be compromised. More specifically, while the performance of our studied pre-trained models can be generally replicated on their originally adopted benchmarks, subtle performance fluctuations can refute the performance comparison results of the studied models in their original papers. For instance, although PLBART reports performance superiority over CodeBERT, our study indicates that it cannot outperform CodeBERT for defect detection and code search. We also find that there hardly exists a dominating pre-trained model and pre-training with multiple objectives (e.g., the encoder-decoder-based models) potentially compromises the pre-training power for individual objectives. Moreover, by comparing with the domain-specific SOTA techniques, we find that the pre-trained models could enable prominent performance on multiple studied tasks, e.g., CodeGPT improves 10.18% and 63.94% on clone detection and code search over the datasets adopted in the original papers of the corresponding SOTA techniques. At last, we propose a new adversarial attack framework for evaluating the robustness of the pre-trained models and demonstrate for the first time that despite the presented excellent performance, the studied models can be easily attacked by a simple random attack approach. Interestingly, even though certain models, e.g., GraphCodeBERT, quite sensitive to noises and attacks under a set of behavioral tests [59]. Such facts altogether can indicate a strong need of an extensive study for the existing NL-PL pre-trained models.

To summarize, our paper makes the following contributions.

- **Dataset.** For the commonly studied tasks, we collect an extensive dataset including a widely-used dataset CodeXGLUE and the datasets from domain-specific techniques such that each task is assigned with common and sufficient datasets for evaluating multiple NL-PL pre-trained models.
- **Study.** An extensive study with general-purpose NL-PL pre-trained models on the proposed datasets which contributes on (1) validating and calibrating performance of the original papers, (2) comprehensive and standardized experimental studies compared to the original papers, (3) adopting adversarial attacks for NL-PL pre-trained models to study their robustness for the first time.
- **Implications.** This work reveals multiple implications including (1) reliable and replicable experiments are essential before releasing pre-trained models since subtle performance fluctuations across multiple runs can easily override their performance comparisons; (2) proposing dominating general-purpose models can be challenging—surprisingly, state-of-the-art encoder-decoder-based models can compromise their encoder components and...
perform even worse than pure encoder-based models for multiple program understand and generation tasks; (3) pre-trained models should be explored more since they tend to outperform conventional SOTA technologies on multiple downstream tasks; (4) as the first study for NL-PL pre-trained model robustness via adversarial attacks, we find that the existing pre-trained models (even the ones with special strategies for improving model robustness) are rather vulnerable.

Note that all our study data, code, and results are publicly available in [2].

2 BACKGROUND

2.1 Program Understanding & Generation Tasks

Automatic program understanding and generation could assist software development and significantly advance the productivity of programmers. Following prior work [43], we now briefly introduce typical program understanding and generation tasks.

2.1.1 Program understanding tasks. Program understanding refers to a set of tasks, e.g., classification and ranking, where models are required to first produce vectorized representations of programs and then perform downstream tasks such as prediction or similarity check.

Defect Detection [84]. This task detects whether a program contains vulnerabilities, e.g., resource leaks and unsafe references.

Clone Detection [50]. This task estimates similarity between two program fragments. Typically it can be realized by binary classification between code pairs and code retrieval respectively.

Code Search [24]. This task matches the semantics between natural language queries and programs.

2.1.2 Program generation tasks. A program generation task requires to produce/generate a sequence of tokens/words either in program languages (e.g., code generation, code repair, and code translation) or natural languages (e.g., code summarization).

Code Summarization [25]. This task generates a short paragraph describing the functionality of a code snippet such that programmers can promptly understand its function.

Code Repair [67]. This task aims to fix program bugs automatically.

Code Translation [9]. This task translates code from one program language to another.

Code Generation [26]. This task generates code by the given natural language description or under the assistance of the predefined policies or artificial intelligence techniques.

2.2 NL-PL Pre-Trained Models

A typical NL-PL pre-trained model refers to pre-training a large model on massive unlabelled corpora by self-supervised objectives, and fine-tuning the model on downstream tasks (i.e., program understanding and generation tasks) with task-specific loss. Many NL-PL pre-trained models [3, 18, 21, 27] have been proposed where their commonly adopted pre-trained schemes are presented in Figure 1. Note that following prior work (e.g., CodeT5 [71] and PLBART [3]), “Encoder” or “Decoder” essentially refers to the Transformer Encoder/Decoder component [69] for simplicity. In particular, the pre-training paradigm contains two stages—one for pre-training and the other for fine-tuning. In the pre-training stage, models are trained by self-supervised objectives on large corpora. Here we take two popular pre-training tasks, Masked language Modeling [15] and Language Modeling [55] for illustration.

Let us now take Figure 2 as an example. Such code snippet will be first flattened by removing all the indents, line breaks, etc., for pre-processing, followed by inserting [CLS] and [SEP] in its front and end position to form an input sequence. Next, in Figure 1a, for the encoder-based models such as CodeBERT [18], GraphCodeBERT [21], and ContraCode [27], they randomly mask parts of the input sequence (e.g., ‘main’, and ‘{‘ are replaced by the [MASK] token) where the objective of their models is to recover the masked tokens (Masked language Modeling). Meanwhile, in Figure 1b, the decoder-based model such as CodeGPT [43] is trained autoregressively, i.e., each generated output will be extended into the original input sequence for subsequent token generation until it reaches an [SEP] (Language Modeling). The encoder-decoder-based models such as CodeT5 [71], CodeTrans [16], CoTExt [53] and PLBART [3] jointly train encoder and decoder for comprehensive modeling of the language. As illustrated in Figure 1c, the encoder first encrypts the polluted (masked) input and injects the features to the decoder which then reads the encoded features to autoregressively generate output tokens along with recovering the masked input. Note that there can be more self-supervised tasks for pre-training models [13, 30]. However, they are not our main focus and thus not presented in this paper. Eventually, we can fine-tune the resulting pre-trained models on small datasets for specific tasks.

Although many NL-PL pre-trained models have been proposed and studied, they are hardly studied on common benchmarks and tasks which can render the evaluation results in their original papers rather biased. Moreover, they fail to comprehensively present performance comparisons with the domain-specific non-pre-trained SOTA techniques for validating their values. At last, limited research work has deeply studied their robustness. While existing work, e.g., CodeXGLUE [43], studies the performance of multiple models, it fails to comprehensively evaluate all of them on universal benchmarks and thus could not fully assess the studied models or draw insights/guidelines for developing/enhancing them.
3 THE EXTENSIVE STUDY

3.1 Subjects and Dataset

3.1.1 Subjects. After CodeBERT was proposed in EMNLP’20, the research on NL-PL pre-trained models (referred to as “pre-trained models” or simply “models” in later sections) has become a hot spot with multiple emerging models. While there can be many potential state-of-the-art pre-trained models to be studied, we choose to filter them for evaluating the models which are representative, publicly available, and designed for a wide range of program understanding and generation tasks for an extensive study. For example, CuBERT [31] only differs from CodeBERT in that CodeBERT additionally adopts the RTD (replaced token detection) [14] pre-training objective. Moreover, TransCoder [60] is only designed for the code translation task. Following prior work [18, 21, 171], we exclude them from our study. As a result, we select eight recently proposed pre-trained models as our studied subjects, i.e., CodeBERT, GraphCodeBERT, ContraCode (all as encoder-based models), CodeGPT (encoder-based model), CodeT5, CodeTrans, CoTexT and PLBART (all as encoder-decoder-based models).

CodeBERT [18]. Following the same architecture of BERT [15], i.e., the transformer-based encoder [69], CodeBERT employs Masked Language Modeling [15] (as in Section 2.2) and Replaced Token Detection [13] (i.e., randomly replacing tokens to train the model for detecting whether the tokens are replaced) as pre-training tasks. GraphCodeBERT [21]. GraphCodeBERT, proposed in ICLR’21, adopts a transformer-based encoder architecture and incorporates code graph structure information (i.e., data flow graph) during pre-training with Edge Prediction (i.e., masking the variables in data flow edges) and Node Alignment (i.e., predicting variable alignment across source code and data flow) as pre-training tasks.

ContraCode [27]. Inspired by contrastive learning [68], ContraCode, proposed in April 2021, augments the training data by performing semantics-preserving transformations. In addition to the Masked Language Modeling objective, it employs InfoNCE [68] loss for contrastive training sample pairs for pre-training.

CodeGPT [43]. CodeGPT, proposed in NeurIPS’21, pre-trains a transformer-based decoder language model GPT [55] on programming languages. Its pre-training process follows the same scheme as illustrated in Figure 1b.

PLBART [3]. PLBART, proposed in NACCL’21, is a transformer-based encoder-decoder inspired by BART [33]. It follows the similar pre-training tasks of BART including Token Masking, Token Deletion, and Token Infilling.

CodeTrans [16]. CodeTrans, proposed in September 2020, is a transformer-based encoder-decoder following the model settings of T5 [56]. It adopts Span Masking [30] (i.e., randomly masking a span of tokens as a whole rather than single element in Masked Language Modeling) and Language Modeling (i.e., auto-regressive language generation) as its objectives to train the model.

CoTexT [53]. CoTexT, proposed in NLP4Prog’21, is also built on the same architecture as T5 [56]. It pre-trains the model with NL-PL data on top of original T5 checkpoints.

CodeT5 [71]. CodeT5, proposed in EMNLP’21, follows the same encoder-decoder architecture as T5 [56]. CodeT5 proposes a novel identifier-aware pre-training task to leverage code-specific structural information as well as a bimodal dual generation pre-training task for augmenting NL-PL alignment.

Table 1 summarizes the features of our studied subjects. While CodeSearchNet which contains six program languages with 2.3M PL-NL function pairs is widely employed as the pre-training corpus, different models actually adopt quite divergent training data subsets. For instance, CodeBERT only uses 2.1M PL-NL function pairs while ContraCode selects 1.8M program functions and only 80K natural language comments. Furthermore, we can clearly observe that in their original papers, they do not adopt the common dataset or tasks for evaluating their effectiveness, e.g., CodeBERT is originally evaluated on CodeSearchNet [24], while ContraCode incorporates varying data sources for type inference and code summarization [4, 22] and also collects a dataset for clone detection. Such facts indicate a pressing need to unify the evaluation protocols for a fair comparison between them.

3.1.2 Datasets. We first aim to adopt a commonly-used benchmark where the studied pre-trained models can be all extensively evaluated. In particular, we adopt the CodeXGLUE [43] dataset, a benchmark constructed by Microsoft for program understanding and generation tasks which collects 14 datasets from previous studies. Note that CodeXGLUE is selected because many of its inclusive datasets have already been adopted for the performance evaluation of multiple studied models in this paper. Moreover, it provides deterministic training/testing data split scheme for all the models to strengthen the fairness of their performance comparison. Specifically, we adopt BigCloneBench [65] and POJ-104 [54] for clone detection, Devign [83] for defect detection, Bugs2Fix [66] for code repair, CodeTrans [43] for code translation, CodeSearchNet-AdvTest [24] for code search, and CodeSearchNet-Java [24] for code summarization. Such datasets are adopted by both our studied subjects in this paper and their original papers and thus can advance the replication of most original evaluations and further extend the study of all the pre-trained models.

Furthermore, for performing a fair comparison between pre-trained models and non-pre-trained SOTA models (illustrated later next section)
in Section 3.3.2), we include 4 datasets adopted for evaluating the SOTA models in their original papers in addition to our CodeXGLUE benchmark. As a result, the pre-trained models and the SOTA models can both be evaluated upon the originally adopted projects and the extended projects. In particular, we add the Reveal1 [10] dataset for defect detection, the POJ-15 [17] dataset for clone detection, the DGSMS-Java [34] dataset for code search, and Rencos-Java [70] for code summarization.

3.1.3 Metrics. Note that many studied pre-trained models have already been evaluated under a set of metrics in their original papers. We then retain all of them in our study for fair performance comparison. Specifically, we adopt Accuracy for defect detection, F1, Precision, Recall, and Mean Average Precision (MAP) for clone detection, and Mean Reciprocal Rank (MRR) for code search. We adopt BLEU-4 for code summarization. For code repair, code translation, and code generation, we adopt BLEU-4, Accuracy, and CodeBLEU. Note that while Accuracy can be adapted for diverse tasks, it essentially denotes the proportion of testing samples whose prediction/generation results are precisely consistent with the target.

3.1.4 Implementation. We obtain the studied models from their corresponding publicly available repositories. We also employ the training/validation/test data splits and fine-tuning procedure for each dataset as in CodeXGLUE. Particularly for the understanding tasks, the models are appended with a Fully-Connected (FC) layer to project the embedding of tokens (e.g. [CLS], the last token) into classes. For the generation tasks, the encoder models (CodeBERT) are appended with a transformer decoder for generating code regressiveively while the decoder (CodeGPT) or encoder-decoder (PLBART) models directly generate code regressiveively. The newly added FC layers or decoders during fine-tuning are first initialized into classes. For the generation tasks, the encoder models (CodeXGLUE [43]) are used as the evaluation metrics. Moreover, for code repair, we adopt two datasets of different sizes (labeled as small with 83830 samples and medium with 65454 samples from [43]) in Table 3 for evaluating how they impact the studied models. We also evaluate bi-directional code translation tasks, i.e., translating from Java to C# and from C# to Java. Meanwhile, by comparing our evaluation results with the corresponding results presented in their original papers, we mark their numerical variations in terms of color gradients as illustrated by “Gap Color Bar” of Table 2, where the color gradients represent the difference magnitude of our study results subtracting the original results. For instance, the F1 score of GraphCodeBERT under clone detection drops 0.72 in our study and is represented as light yellow in Table 2. Moreover, the BLEU-4 score of CodeTrans under code summarization is increased by 0.20 in our study and represented as light green in Table 3. Note that the rest unmarked statistics refer to our extended evaluation results which are not included by their corresponding original papers.

We find that most of our results can be consistent with the original results. For instance, the F1 score of GraphCodeBERT under clone detection drops 0.72 in our study and is represented as light yellow in Table 2. Moreover, the BLEU-4 score of CodeTrans under code summarization is increased by 0.20 in our study and represented as light green in Table 3. Note that the rest unmarked statistics refer to our extended evaluation results which are not included by their corresponding original papers.

Table 2: Evaluation results on program understanding tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>Defect Detection</th>
<th>Clone Detection</th>
<th>Code Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>Precision</td>
<td>Recall</td>
<td>F1</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>63.68</td>
<td>95.94</td>
<td>90.94</td>
</tr>
<tr>
<td>GraphCodeBERT</td>
<td>63.35</td>
<td>95.80</td>
<td>90.96</td>
</tr>
<tr>
<td>ContraCode</td>
<td>58.42</td>
<td>95.04</td>
<td>96.93</td>
</tr>
<tr>
<td>CodeGPT</td>
<td>63.49</td>
<td>95.31</td>
<td>96.93</td>
</tr>
<tr>
<td>PLBART</td>
<td>60.77</td>
<td>96.00</td>
<td>98.21</td>
</tr>
<tr>
<td>CodeTrans</td>
<td>63.03</td>
<td>90.56</td>
<td>95.40</td>
</tr>
<tr>
<td>CoTeXt</td>
<td>61.85</td>
<td>90.96</td>
<td>97.74</td>
</tr>
<tr>
<td>CodeTS</td>
<td>63.62</td>
<td>92.44</td>
<td>95.60</td>
</tr>
</tbody>
</table>

3.2 Research Questions

We investigate the following research questions in our study:

- **RQ1**: How do pre-trained models perform for program understanding and generation tasks? For this RQ, we extensively study the performance evaluation between all the adopted pre-trained models upon the CodeXGLUE benchmark [43].
- **RQ2**: How do pre-trained models perform against non-pre-trained models? For this RQ, we evaluate and compare the performance between pre-trained models and domain-specific non-pre-trained state-of-the-art (SOTA) models on multiple tasks.
- **RQ3**: Are the pre-trained models robust? For this RQ, we apply semantics-preserving adversarial attack techniques to investigate the robustness of the studied models.

3.3 Results and Analysis

3.3.1 **RQ1**: How do the pre-trained models perform for program understanding and generation tasks? While the studied pre-trained models investigate diverse program understanding and generation tasks, we determine to evaluate the tasks which have been studied by more than one of our studied subjects, i.e., the listed tasks in Section 2.1, under the fine-tuning process provided in prior work CodeXGLUE [43] for extensively studying and comparing their performance. Note that considering the model randomness, we run all our experiments for 5 runs as recommended by prior work [52] and present the average results, while all our studied pre-trained models do not clearly indicate their number of runs in their original papers.

Tables 2 and 3 demonstrate the average experimental results in program understanding and generation tasks respectively on top of benchmark CodeXGLUE. Specifically, we adopt dataset BigCloneBench for evaluating clone detection as a classification problem [8, 17, 43], i.e., determining whether a pair of code snippets are cloned, where Precision, Recall, and F1 are used as the evaluation metrics. We also adopt dataset POJ-104 for evaluating clone detection as a retrieval problem [43, 51], i.e., retrieving the cloned samples of a given code snippet from the whole testing set, where MAP is used as the evaluation metric. Moreover, for code repair, we adopt two datasets of different sizes (labeled as small with 83830 samples and medium with 65454 samples from [43]) in Table 3 for evaluating how they impact the studied models. We also evaluate bi-directional code translation tasks, i.e., translating from Java to C# and from C# to Java. Meanwhile, by comparing our evaluation results with the corresponding results presented in their original papers, we mark their numerical variations in terms of color gradients as illustrated by “Gap Color Bar” of Table 2, where the color gradients represent the difference magnitude of our study results subtracting the original results. For instance, the F1 score of GraphCodeBERT under clone detection drops 0.72 in our study and is represented as light yellow in Table 2. Moreover, the BLEU-4 score of CodeTrans under code summarization is increased by 0.20 in our study and represented as light green in Table 3. Note that the rest unmarked statistics refer to our extended evaluation results which are not included by their corresponding original papers.

We find that most of our results can be consistent with the original results. For instance, the F1 score for GraphCodeBERT under clone detection drops 0.72 in our study and is represented as light yellow in Table 2. Moreover, the BLEU-4 score of CodeTrans under code summarization is increased by 0.20 in our study and represented as light green in Table 3. Note that the rest unmarked statistics refer to our extended evaluation results which are not included by their corresponding original papers.

However, multiple performance comparison results in our study can be reversed compared with their original papers. Specifically, Figure 3 presents the average F1 and min-max variations across 5 runs of code detection under BigCloneBench. We notice that...
the performance discrepancies of all the executions cause non-negligible impact on performance comparison among different models. For example, CodeGPT incurs the largest performance variance, i.e., it can outperform all the other pre-trained models in one run and underperform them in another run. Moreover, although PLBART reports 0.7 F1 advantage over CodeBERT in its original paper, the min-max gap of PLBART in our study, i.e., 0.84 in Figure 3, may easily make such results untenable. Additionally, Table 2 presents that the defect detection accuracy of CodeBERT and PLBART is 63.68 and 60.77 respectively in our study while 62.08 and 63.18 respectively in the PLBART paper. Although such fluctuations appear to be minor/limited, they can easily reverse the performance comparison results and potentially compromise the strength of PLBART (and some other models).

Finding 1: The performance of all studied models can overall be replicated on their original benchmarks. However, subtle performance fluctuations can result in the untenable findings in their original papers, e.g., PLBART fails to outperform CodeBERT in defect detection and clone detection.

Program Understanding Tasks. We observe from Table 2 that the encoder-based CodeBERT and GraphCodeBERT can generally provide the optimal performance among all studied pre-trained models. Specifically, GraphCodeBERT outperforms CodeT5 by 24.91% and 62.11% for code search and clone detection under POJ-104 respectively. On the other hand, for clone detection under BigCloneBench and defect detection, the encoder-decoder-based models achieve similar performance with CodeBERT and GraphCodeBERT, i.e., around 1% performance variance. Note that such results conceptually correspond to the findings in the research domain of natural language models [33, 56]—while encoder-based models can in general outperform encoder-decoder-based models in language understanding, their performance difference in certain tasks (e.g., Stanford Sentiment Treebank and Quora Question Pairs [33]) can be rather limited.

Finding 2: Encoder-based pre-trained models can achieve similar or even superior performance over encoder-decoder-based models on specific program generation tasks.

Interestingly, Tables 2 and 3 both present that for the tasks evaluated via multiple metrics, the studied models incur quite fluctuating performance under all the adopted metrics. Especially for code repair, the optimal performance under the 3 metrics are achieved by 3 different models. Also, for program understanding tasks, CodeBERT, the first of its kind, still outperforms later proposed CodeT5 and PLBART which declare their performance gain on CodeBERT in their original papers. We observe from Table 3 that in general, the encoder-decoder-based pre-trained models can outperform other models on most program generation tasks. Specifically, CodeT5 and CodeTrans enable the optimal performance on code summarization, translation, and generation. Interestingly, we also find that on specific generation tasks, encoder-decoder-based models barely outperform encoder-based models. Specifically, for code repair, CodeBERT and GraphCodeBERT enable the optimal CodeBLEU under the Small and Medium dataset respectively (i.e., 78.73 and 88.38). For code generation, CodeBERT and GraphCodeBERT slightly outperform CodeTrans and CoText under BLEU-4, i.e., 34.20 and 34.82 v.s. 33.45 and 32.14. For code summarization, GraphCodeBERT performs similarly with PLBART and CoText, i.e., 19.10 v.s. 19.02 and 19.37.

We can summarize from such results above that encoder-based pre-trained models can achieve similar or even superior performance over encoder-decoder-based models on specific program generation tasks (e.g., code repair), i.e., the encoder-based models can be potentially as general-purpose as encoder-decoder-based models. Such fact seems to be overlooked by previous studies [33, 56] which believe the encoder-decoder architecture is optimal for general-purpose pre-trained models. Note that for decoder-based CodeGPT, it fails to enable optimal performance on any task. We thus infer that only adopting decoder architecture is inadequate for building general-purpose pre-trained models.

Table 3: Evaluation results on program generation tasks

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Code Summarization</th>
<th>Code Repair</th>
<th>Code Translation</th>
<th>Code Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Acc(%)</td>
<td>F1</td>
<td>Acc(%)</td>
<td>Acc(%)</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>BLEU-4</td>
<td>BLEU-4</td>
<td>BLEU-4</td>
<td>BLEU-4</td>
</tr>
<tr>
<td>Small</td>
<td>74.36</td>
<td>77.13</td>
<td>80.56</td>
<td>80.55</td>
</tr>
<tr>
<td>Medium</td>
<td>77.13</td>
<td>80.56</td>
<td>80.55</td>
<td>80.55</td>
</tr>
<tr>
<td>CoText</td>
<td>74.36</td>
<td>77.13</td>
<td>80.56</td>
<td>80.55</td>
</tr>
</tbody>
</table>

Figure 3: Min-max F1 variations of clone detection under BigCloneBench
typically reflects rich information to understand a sequence and is favored for program understanding tasks. On the other hand, encoder-decoder-based models only take advantage of the informative encoder representations to advance its decoder to generate fluent and meaningful sequences, i.e., they do not fully explore the potential power of their program understanding capabilities. Consequently, the performance of encoder-decoder-based models on program understanding tasks are hindered by the generation objective, i.e., enhancing program generation performance is in fact at the cost of program understanding tasks. Note that such trade-off is also well recognized in the NLP domain [33, 56]. Accordingly, we derive that it is challenging or even impractical to develop an almighty model which rules all the tasks with current pre-training objectives, e.g., Masked Language Modeling and auto-regressive loss.

Finding 3: There exists no dominating pre-trained models. One main reason is that the current SOTA encoder-decoder models largely overlooked the fact that different training objectives (for program understanding and generation) can potentially compromise each other. We call for future research on more inclusive encoder-decoder models.

3.3.2 RQ2: How do pre-trained models perform against non-pre-trained models? Pre-trained models are expected with prominent performance because they can effectively capture rich knowledge (e.g., syntax and semantics) from massive pre-training data compared with non-pre-trained models (e.g., training corpora size by millions for pre-trained models in Table 1 vs. by tens of thousands for non-pre-trained models in Table 4). Such data are implicitly encoded via a huge number of parameters, which can eventually facilitate a variety of downstream tasks [74]. For validating such initiative, it is essential to compare the performance between the pre-trained models and the domain-specific non-pre-trained SOTA approaches. To this end, we search the most recently published papers in top academic venues for identifying all the tasks which are presented that they can outperform a number of well-known previously published techniques. Accordingly, we also attempt for generalized SOTA techniques such that the performance comparison can be limited within the identical scope. Eventually, we adopt the following domain-specific SOTA techniques.

Reveal [10]: Chakraborty et al. conduct a study over deep-learning-based defect detection in TSE’21. They investigate the deficiency of the existing defect detection datasets and propose a more refined dataset. Meanwhile, they propose a graph neural network (GNN) approach Reveal which leverages program structure information, i.e., AST and CFG, and significantly outperforms the previous SOTA approaches in both precision (by 33.57%) and recall (by 128.38%).

FCDetector [17]: Fang et al. propose FCDetector to extract syntax and semantic information from source code for clone detection in ISSTA’20. In particular, FCDetector joins different functions in programs through caller-callee relationships and then applies its own fusion embedding techniques[17] to learn hidden syntactic and semantic features of the whole programs. It can significantly outperform previous SOTA approaches. Meanwhile, they also construct a smaller code clone dataset (with 15 label types) based on P0J-104 which is named as P0J-15 in this paper.

DGMS [37]: Ling et al. propose a code search approach DGMS in TKDD’21. DGMS leverages GNN and program structure information and also extracts structure information of natural language query for natural code search. Their evaluation results demonstrate that the DGMS model significantly outperforms SOTA models in MRR (up to 31.53%). Furthermore, we also adopt their code search dataset as one of our extended datasets for a fair comparison.

Rencos [79]: Rencos is a retrieval-based code summarization approach proposed in ICSE’20 which can take advantage of both neural and retrieval-based techniques. Previous studies [62, 79] have shown that Rencos outperforms other non-pre-training SOTA approaches. Meanwhile, we also adopt their code summarization dataset as one of our extended datasets for a fair comparison.

To the best of our knowledge, such approaches are so far the latest and highly regarded work in their domains. On the other hand, we could not find a proper non-pre-trained model for code generation, code repair, and code translation. In particular, the SOTA code generation approach NL2Code [73] relies on external knowledge, e.g., API documentation of specific program language, to generate code, and the SOTA code repair approaches [20, 44] require precise fault localization [40, 80] information, and thus can hardly be extended to the CodeXGLUE dataset. While we can trace back to earlier domain-specific techniques, i.e., CONCODE [26], CODE-TRANS [67], which adopt RNN-based seq2seq models, previous studies [21, 28, 60] have shown that pre-trained models using the transformer architecture can largely outperform them in program generation tasks. For code translation, the latest research work [60] also demonstrates that the pre-trained models can outperform non-pre-trained domain-specific techniques.

Table 4 presents the performance comparison results of pre-trained and non-pre-trained SOTA models. Note that we adopt P0J-15 for evaluating clone detection as a classification problem corresponding to BigCloneBench in Table 2, while P0J-104 is used for retrieval-based clone detection. By comparing the clone detection results under BigCloneBench, we observe that the performance of CodeBERT and GraphCodeBERT cannot be generalized to the extended P0J-15 dataset. In particular, GraphCodeBERT achieves the optimal MAP of 89.97 under dataset P0J-104 and achieves 96.38 F1 under dataset BigCloneBench for clone detection. However, its F1 score drops to 66.80 under the P0J-15 dataset. Such dramatic performance degradation also occurs for CodeBERT (from 96.22 to 55.46), ContraCode (from 95.96 to 50.47), and Code-Trans (from 92.93 to 66.73). The above results may be due to the fact that the hyperparameters adopted directly from their original papers are not all optimal for our dataset, e.g., it is more suitable for above models under BigCloneBench rather than P0J-15. We thus infer that even for the pre-trained models which are supposedly general-purpose for various benchmarks may incur severe performance degradation, i.e., the pre-trained models are data-dependent.

Finding 4: General-purpose pre-trained models can incur dramatic performance variance.

We can observe that for most tasks, the pre-trained models can significantly outperform the domain-specific SOTA models. In particular, for defect detection, CodeBERT and CoText achieve the
We assume such performance may be caused since Rencos can while, for clone detection under POJ-15 may cheat the anti-defect system via semantics-preserving edit pre-trained models especially when they can be easily manipulat- under different datasets which implies the potential model robust- that pre-trained models can enable quite inconsistent performance 3.3.3 RQ3: Are the pre-trained models robust? Finding 4 reveals optimal Accuracy of 63.68 and 90.72 respectively under CodeXGLUE and Reveal1. Such performance can outperform the SOTA technique Reveal by 11.78% and 4.00% respectively. For clone detection under POJ-104 (illustrated in Section 3.3.1), GraphCodeBERT achieves a MAP of 89.97 and largely exceeds non-pre-trained SOTA model FCDetector which obtains 27.49. Meanwhile, for clone detection under POJ-15 with the same setting as the FCDetector paper, CodeGPT and PLBART strongly outperform FCDetector by achieving 100 in terms of F1, Precision, and Recall. Note that POJ-15 (introduced in FCDetector) is a smaller and simpler dataset compared with BigCloneBench, i.e., 1510 vs. 9126 code fragments. Moreover, POJ-15 incurs a data leakage issue [61], i.e., part of the testing samples can be seen during training, while BigCloneBench incurs no such issue [43]. As a result, CodeGPT and PLBART can achieve excellent performance on POJ-15 while others cannot due to the data-dependency issue (Finding 4). Note that for code summarization, SOTA Rencos outperforms GraphCodeBERT by 35.64% under the dataset adopted in its original paper and underperforms it by 19.78% in the CodeXGLUE dataset. We assume such performance may be caused since Rencos can be quite data-dependent. Specifically, when summarizing target programs, Rencos first retrieves similar code snippets from the training dataset, and regards them as part of the input to its adopted neural network for code summarization. Therefore, it can achieve superior performance in the dataset which contains substantial duplicate samples across training and testing datasets.

To summarize, we infer that pre-trained models generally out- perform non-pre-trained SOTA models in program understanding tasks. Since they enable powerful performance gain, establishing domain-specific SOTA techniques by gradually upgrading the existing ones can take a long way to go. Therefore, we recommend researchers to make more efforts in studying pre-trained models.

Finding 5: Pre-trained models could be more promising than non-pre-trained models and more research efforts should be devoted to this direction.

<table>
<thead>
<tr>
<th>Task</th>
<th>Defect Detection</th>
<th>Clone Detection</th>
<th>Code Search</th>
<th>Code Summarization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>CodeXGLUE</td>
<td>Reveal Dataset</td>
<td>POJ-104</td>
<td>POJ-15</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>63.68</td>
<td>89.57</td>
<td>86.78</td>
<td>51.55</td>
</tr>
<tr>
<td>GraphCodeBERT</td>
<td>63.35</td>
<td>90.15</td>
<td>89.97</td>
<td>50.23</td>
</tr>
<tr>
<td>ContraCode</td>
<td>58.42</td>
<td>89.78</td>
<td>45.65</td>
<td>50.61</td>
</tr>
<tr>
<td>CodeGPT</td>
<td>63.89</td>
<td>89.79</td>
<td>71.22</td>
<td>100.00</td>
</tr>
<tr>
<td>PLBART</td>
<td>60.77</td>
<td>89.09</td>
<td>62.26</td>
<td>100.00</td>
</tr>
<tr>
<td>CodeTrans</td>
<td>63.03</td>
<td>89.62</td>
<td>61.88</td>
<td>50.08</td>
</tr>
<tr>
<td>CoText</td>
<td>61.48</td>
<td>90.72</td>
<td>58.72</td>
<td>50.40</td>
</tr>
<tr>
<td>CodeT5</td>
<td>63.62</td>
<td>85.97</td>
<td>55.50</td>
<td>53.76</td>
</tr>
</tbody>
</table>

Not Pre-trained SOTA

<table>
<thead>
<tr>
<th>Task</th>
<th>Defect Detection</th>
<th>Clone Detection</th>
<th>Code Search</th>
<th>Code Summarization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>CodeXGLUE</td>
<td>Reveal Dataset</td>
<td>POJ-104</td>
<td>POJ-15</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>56.97</td>
<td>87.23</td>
<td>27.49</td>
<td>95.69</td>
</tr>
</tbody>
</table>

Algorithm 1: CodeAttack

**Input**: P, k, n, search, transformation

**Result**: P'

```plaintext
1. best_P' ← P;
2. sites ← Identify the sites that can be edited in P by AdoCG;
3. k_sites ← search(sites, k);
4. for site in k_sites do
   5. P_list ← transformation(site, best_P', n);
   6. best_P' ← evaluate each P' in P_list + best_P' and get the best_P';
   7. if best_P' can fool the model then
      8. return best_P';
   end
end
10. return best_P'
```

However, to the best of our knowledge, there exists no adversarial attack approach specifically designed for NL-PL pre-trained models before this study. In particular, in programming language area, while the previous attacking approaches [63, 76] for programming languages employ one-hot embedding [76], all existing NL-PL pre-trained models adopt continuous vector embedding [48] which renders the one-hot embedding attacking inapplicable. As for adversarial attacks in natural languages, models are allowed to modify every word in sentence. Simply following such unconstrained operation in NL-PL pre-trained models may edit key words of programs, e.g., “for” and “while”, and thus possibly break program semantics and fail program executions. To safely conduct adversarial attacks which preserve program semantics and syntax for pre-trained models, we propose a new framework based on AdvCG [63] (originally designed for non-pre-trained models to search and find sites which are safe to be modified in programs) and attacking approaches from NLP [19, 29, 35, 45, 46] (adversarial attack approaches that modify the sites with optimal perturbations). As illustrated in Algorithm 1, for a program P, we initialize the best perturbed P' as P (Lines 1). AdvCG is employed to analyze the code and identify sites that can be safely edited without changing the program semantics, e.g.,

(demonstrated in Table 5) for avoiding additional code review process. Such tricks may increase the threats of program crashes or deadlocks. The cheating process can be considered as adversarial attack to the pre-trained models [63, 76]. Therefore, we determine to apply semantics-preserving adversarial samples to attack the pre-trained models for inspecting their robustness.
An Extensive Study on Pre-trained Models for Program Understanding and Generation

Table 5: Evaluating robustness under defect detection and code summarization

<table>
<thead>
<tr>
<th>Performance under adversarial attack</th>
<th>Defect Detection</th>
<th>Code Summarization</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeBERT</td>
<td>83.49</td>
<td>19.48</td>
</tr>
<tr>
<td>GraphCodeBERT</td>
<td>81.33</td>
<td>18.33</td>
</tr>
<tr>
<td>ContraCode</td>
<td>64.81</td>
<td>22.12</td>
</tr>
<tr>
<td>CodeGPT</td>
<td>64.81</td>
<td>22.12</td>
</tr>
</tbody>
</table>

Table 6: Attack approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Search Method</th>
<th>Transformation Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAE [49]</td>
<td>Word Importance Rank</td>
<td>Word Swap Language Model BAE</td>
</tr>
<tr>
<td>LSH [46]</td>
<td>Word Importance Rank</td>
<td>Word Swap WordNet</td>
</tr>
<tr>
<td>TextFooler [29]</td>
<td>Word Importance Rank</td>
<td>Word Swap Embedding</td>
</tr>
<tr>
<td>Random</td>
<td>Random</td>
<td>Random</td>
</tr>
<tr>
<td>WIR-Random</td>
<td>Word Importance Rank</td>
<td>Random</td>
</tr>
<tr>
<td>Random+</td>
<td>Random+</td>
<td>Random+</td>
</tr>
</tbody>
</table>

local variables and function parameters (Lines 2). Then, a search method is applied to select up to K sites which can potentially lead to optimal perturbation (Line 3). For each resulting candidate site, a transformation method, which replaces the original token in the site with another word, is employed n times and results in n perturbations of the program P (Lines 4-5). Such n perturbations are evaluated and the one which can best fool the pre-trained model is selected as new program $P'$. The genetic algorithm is slightly modified to be more robust under iterative operations (Lines 6-9). Note that the program semantics and syntax are ensured to be unchanged since all the modifications are performed only on safe sites identified by AdvCGR which is widely used by attack approaches [57, 76] for programming languages.

Table 6 summarizes the details of the search and transformation methods, as described in TextAttack [49]. For the search methods, the Search Method algorithm is employed to rank each site according to the output magnitude difference before and after renaming the site to "UNK" token, while Genetic Algorithm [45] finds the optimal combination of sites and transformations by mutation and crossover of different perturbations. For the transformation methods, all of them aim to find a suitable token to replace the original one in a given site. Word Swap Embedding [49] returns a synonym via extra embedding model. Word Swap WordNet [46] returns a synonym via a WordNet model. Word Swap Language Model BERT [35] returns a new token generated by an extra BERT model. An attack approach can be composed by different search methods and transformation methods, as described in TextAttack [49]. In addition, we also design a simple random attack approach (namely Random approach) by randomly selecting attack sites and then renaming each selected site with a random new word. Note that the random selection is also semantics-preserving following Lines 2-3 of Algorithm 1 as the safe sites have been selected and we only randomly select/replace them. We also design a WIR-Random approach (introduced later) to further study the effectiveness of different search methods.

In addition, CodeT5 and CodeGPT are relatively more robust than other pre-trained models. For instance, the performance of CodeT5 drops 58.93% for defect detection under the Random attack, less than 69.98% of CodeTrans and 66.87% of CoText which both build on the same T5 architecture. The superiority of CodeT5 could be caused by its unique identifier tagging pre-training task, i.e., identifying whether a code token is an identifier and advancing the model interpretability [32]. Specifically, we extract the attention weights of the first layer for the GraphCodeBERT model where such weights can indicate how much attention GraphCodeBERT pays to

Finding 6: We demonstrate for the first time that NL-PL pre-trained models are not robust. They are highly vulnerable against semantics-preserving adversarial samples.
We then compare the performance variance across different attack approaches. Interestingly, we find that the Random approach can achieve the optimal attack performance in 6 of 8 studied models for defect detection, while for the other models, it also enables rather strong performance, e.g., 6.47 vs. 4.98 on GraphCodeBERT.

For code summarization, the Hard-Label approach outperforms the Random approach on several subjects, e.g., the BLEU-4 results of CodeBERT and GraphCodeBERT on code summarization (2.30 and 1.31 of the Random approach vs. 1.69 and 0.58 of the Hard-Label approach). We observe that the average query number, i.e., the number of searching the optimal perturbation, of the Hard-Label approach largely exceeds the Random approach (i.e., 925.51 and 814.53 vs. 130.46 and 156.51 in terms of defect detection and code summarization). Thus, we hypothesize that the superiority of the Hard-Label approach over the Random approach is caused by generating and evaluating more perturbations. Accordingly, we attempt to improve the Random approach by adopting a larger \( n \) in Algorithm 1 which allows more perturbations for each site. In this way, the Random approach outperforms all the other attack approaches in 14 of 16 subjects as in "Random+" of Table 7.

Finding 8: A simple random attack approach can already be rather powerful for attacking NL-PL pre-trained models.

We further analyze the difference of the performance impact between the search and transformation method. For the transformation method, the NLP attack approaches tend to preserve the semantics at the natural language level by adopting synonyms for substitution, i.e., selecting the semantically similar words with extra embedding model. On the other hand, program semantics can be preserved even when the variable names are changed. Therefore, unnecessary restrictions can decrease the transformation method effectiveness. Subsequently, to analyze the impact of the search method, we build WIR-Random approach that combines the Word Importance Rank (WIR) search method and Random transformation method. Then, by comparing the WIR-Random result in Table 7 and the Random result in Table 5, we find that the WIR search method enables limited advantage over the Random search method. From such result we can infer that transformation methods incur a greater impact on attack performance than search methods. While in the NLP domain, the search methods may incur more performance significant impact as discussed in [77], because there are more attackable sites in natural statements than programs.

Finding 9: Transformation methods tend to impact more on attack performance than search methods for NL-PL pre-trained models.

4 IMPLICATIONS AND DISCUSSIONS

Our study reveals the following important practical guidelines for future research on pre-trained models.

Pre-training for code is promising. Our study results show that the pre-trained code models can outperform the non-pre-trained SOTA techniques upon almost all the evaluation tasks. Therefore,
we strongly recommend applying pre-trained models for various program understanding and generation tasks.

**More rigorous evaluations are needed.** Considering the stochastic nature of neural networks and the small performance deviations between different models on some tasks (Finding 1), we recommend researchers to replicate experiments when validating their techniques to prevent over-claiming their advantages.

**Multi-objective v.s. single-objective pre-training**. Our study demonstrates that it can be rather challenging for researchers to propose general-purpose pre-trained models for dominating both program understanding and generation tasks. Specifically, it is hard because the models targeting multi-objective optimization tasks can potentially compromise their efficacy compared with the ones targeting single-objective optimization task.

**Model robustness shall be further explored.** Our experimental results show that the existing pre-trained models encounter significantly decreased performance even under random attacks. Moreover, so far, simply incorporating program semantics fails to enhance their robustness. Therefore, we recommend researchers to put more efforts to enhance robustness of pre-trained models.

## 5 Threats to Validity

### Threats to Internal Validity

The threats to internal validity mainly lie in the potential bugs in our implementation. To reduce such threats, we directly obtain the original source code from the GitHub repositories of the studied techniques. Also, we adopt the same hyperparameters as their original papers. The authors also carefully reviewed the experimental scripts to ensure the correctness. Meanwhile, while re-pretraining a model can be essentially helpful for our study, most of the studied models do not provide their pre-training code and dataset which prevent us from realizing such options. To reduce such threat, we design three research questions for evaluating their provided pre-trained models from multiple dimensions for assessing their characteristics.

### Threats to External Validity

The threats to external validity mainly lie in the benchmark and techniques adopted in this study. To reduce such threats, we not only use the CodeXGLUE benchmark studied in many original pre-trained model papers, but also include more datasets to construct an extended benchmark. Meanwhile, through an exhaustive literature review, we believe that the pre-trained models adopted in this study are sufficiently representative and influential in this domain, and we also assure that the non-pre-trained models adopted in this study are the SOTA techniques for their respective tasks.

### Threats to Construct Validity

The threats to construct validity mainly lies in the adopted metrics in our evaluations. To reduce such threats, following the prior CodeXGLUE work, we adopt multiple widely-used metrics to evaluate the performance of the studied approaches, i.e., **Accuracy**, **F1**, **Precision**, **Recall**, **MAP**, **MRR**, **BLEU-4**, and **CodeBLEU**. Meanwhile, for some tasks, we apply more than one metrics to evaluate their performance. To further reduce the threats, we also perform case studies to validate our hypothesis.

## 6 Related Work

Pre-trained language models have been shown to be effective in NLP [15, 33, 39, 55]. BERT [15] is one of the most representative works of its kind, a multilayer transformer-based encoder, which achieves SOTA results on several natural language understanding (NLU) tasks. By investigating the BERT, Liu et al. [39] find that BERT was significantly undertrained, and a fine-grained selection of hyperparameters can dramatically improve the performance of BERT. However, noticing that BERT-based pre-trained language models are not suitable for natural language generation (NLG) tasks, the decoder structure was introduced (e.g., BART [33], T5 [56]) to cope with both NLG and NLU tasks. With the remarkable success of pre-trained models in NLP, multi-modal pre-trained models have also been proposed. Such models can learn implicit alignments between different modal inputs and thus can be used for cross-domain tasks, e.g., ViLBERT [42] for language and image, VideoBERT [64] for language and video, and SpeechT5 [7] for language and speech.

In recent years, researchers have explored various techniques in the intersection of software engineering and machine learning [5, 8, 23, 36, 41, 70, 72, 78, 81, 82]. Program understanding and generation bring together various program-related tasks, such as defect detection, clone detection, code search, code summarization, code generation, etc. Such tasks can be well transformed into machine learning problems. For instance, Alon et al. [5] applied neural network techniques for generating distributed representations of code. Bui et al. [8] proposed InferCode to learn program representations by predicting subtrees sampled from program ASTs. Hoang et al. [23] proposed CC2Vec to learn a representation of commits by commit messages and commit code changes. Meanwhile, pre-trained language models have also been applied to various program understanding and generation tasks. In addition to our studied models, Kanade et al. [31] apply BERT model to learn the contextual embedding of source code. Liu et al. [38] pre-trained a BERT model for code completion. Lachaux et al. [60] pre-trained a transformer on source code from open source projects for code translation, and such a model can achieve high accuracy under code translation task between C++, Java, Python functions. Furthermore, Chen et al. [12] proposed an enormous program language GPT-3 model with up to 12 billion parameters and introduced its commercial application GitHub Copilot. In this paper, we conduct the first comprehensive study on multiple SOTA pre-trained models upon universal benchmarks with in-depth analysis to advance future research. Note that CodeXGLUE [43] with its leaderboard mainly presents a benchmark and evaluates baselines with limited analytical results and hardly evaluates all studied models simultaneously on universal benchmarks.

## 7 Conclusions

In this study, we have extensively investigated the effectiveness and limitations of NL-PL pre-trained models for program understanding and generation tasks. We first discover the performance fluctuation of different pre-trained models over different tasks and datasets, which indicates that it can be challenging to propose an almighty pre-trained model across task types and it is essential for reliable experiments to demonstrate the superiority of proposing new models. Furthermore, we also validate the superiority of pre-trained models over conventional/previous SOTA methods on different downstream tasks. Finally, we perform the first study for NL-PL pre-trained model robustness via adversarial attacks and find that
the existing pre-trained models are rather vulnerable, e.g., they can be easily attacked by a simple random attack approach, and current strategies for improving the robustness of pre-trained code models have limited effectiveness. Therefore, researchers should make more efforts on proposing integration schemes of additional information with pre-training.

8 ACKNOWLEDGEMENT

This work is partially supported by the National Natural Science Foundation of China (Grant No. 61902169), Guangdong Provincial Key Laboratory (Grant No. 2020B121201001), and Shenzhen Peacock Plan (Grant No. KQTD201611251455551). This work is also partially supported by National Science Foundation under Grant Nos. CCF-2131943 and CCF-2141474, as well as the Google Research Scholar Program and Kwai Inc. Yuqun Zhang is also deeply grateful for the birth of his little girl who brings all the joy, love and inspirations.

REFERENCES


