2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM) | 978-1-6654-9609-4/22/$31.00 ©2022 IEEE | DOI: 10.1109/SCAM55253.2022.00014

An Empirical Study of Code Smells in Transformer-based Code Generation Techniques

Mohammed Latif Siddiq*, Shafayat H. Majumder†§, Maisha R. Mim†§, Sourov Jajodia†, Joanna C. S. Santos*
*Department of Computer Science and Engineering, University of Notre Dame, USA
†Department of Computer Science, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh
msiddiq3@nd.edu, {1705080, 1705060, 1705072}@ugrad.cse.buet.ac.bd, joannacss@nd.edu

***Abstract*—Prior works have developed transformer-based language learning models to automatically generate source code for a task without compilation errors. The datasets used to train these techniques include samples from open source projects which may not be free of security flaws, code smells, and violations of standard coding practices. Therefore, we investigate to what extent code smells are present in the datasets of coding generation techniques and verify whether they leak into the output of these techniques. To conduct this study, we used Pylint and Bandit to detect code smells and security smells in three widely used training sets (CodeXGlue, APPS, and Code Clippy). We observed that Pylint caught 264 code smell types, whereas Bandit located 44 security smell types in these three datasets used for training code generation techniques. By analyzing the output from ten different configurations of the open-source fine-tuned transformer-based GPT-Neo 125M parameters model, we observed that this model leaked the smells and non-standard practices to the generated source code. When analyzing GitHub Copilot's suggestions, a closed source code generation tool, we observed that it contained 18 types of code smells, including substandard coding patterns and 2 security smell types.**

***Index Terms*—code generation, code smell, security smell, transformer, pre-trained model, GitHub copilot**

I. Introduction

Code generation techniques aim to automatically generate functional code based on prompts [1]. These prompts can include textual descriptions (comments), code (*e.g.*, function signatures, expressions, variable names, *etc.*), or a combination of these (*i.e.*, free text and code). As a result, developers can write an initial code and/or comment and rely on these tools to generate the remaining code, which saves them time and helps to speed up the software development process.

Recently, there has been an uptick of machine-learning-based techniques that uses Natural Language Processing (NLP) techniques trained with a large amount of code snippets (instead of plain text) to generate code from user prompts [2]. These Large Language Learning Models (LLMs) are trained with a large dataset of source code snippets and fine-tuned for a wide range of software engineering applications, such as automated code completion [3]–[5], summarization [6], documentation [7], and generation [8], [9]. LLMs are mainly based on an *attention-based transformer model* [10] with different parameters to catch the context of natural language.

§These authors equally contributed to this work.

Although a good code generation tool can help programmers in reducing development efforts [11], the code snippets used to train these techniques are typically taken from open-source repositories. These code samples used for training may not have been adequately vetted in terms of following coding standards, implementing proper design decisions, and using secure (defensive) coding practices [12], [13]. Therefore, these datasets may contain *code smells* that may affect the quality of the code generated by these techniques [14].

Code smells are symptoms that may indicate the system has flaws [15]. For example, a class with an unusually long method may indicate poor design choices. Although code smells may not directly affect the software's functionality, they can introduce long-term maintainability issues and technical debt [13]. They can also compromise the system's security [16]. This subset of smells, commonly referred to as *security smells*, is not necessarily an actual vulnerability, but it can open the door for developers to make mistakes that lead to security flaws that are exploitable by attackers [17], [18].

Although prior research has studied the presence of code smells in machine learning [19] and data science projects [20] (*i.e.*, in the actual code that implements the models), they have not studied the training set and the output of these projects. Moreover, prior research focused on verifying whether the generated code is *functionally correct* (*i.e.*, it implements what the user expected) [2], [11], [21] but not the *quality* of the generated code. More recent research investigated whether GitHub Copilot (a closed source code generation tool) can generate vulnerable code [22] or whether it is as bad as humans in generating insecure code [23]. However, we currently lack an in-depth understanding of the quality of training datasets and whether code smells can potentially leak into the code generated by these models.

Therefore, we conduct **a large-scale empirical study of code smells in the training sets of transformer-based code generation models for Python and investigate the leakage of these harmful patterns to the output**. To conduct this study, we retrieved three open-source datasets (CodeXGlue [24], APPS [25], and Code Clippy [26]) commonly used for training Python code generation techniques and verified to what extent they contain code smells. Furthermore, we investigated whether the code generated by transformer-based models can

2470-6892/22/$31.00 ©2022 IEEE
DOI 10.1109/SCAM55253.2022.00014

contain code smells. For this investigation, we computed the code smells in the outputs generated by GPT-Code-Clippy [27] and GitHub Copilot [28], an open-source and closed-source code generation tool, respectively. Our scripts used in this research are publicly available in our repository: https://github.com/s2e-lab/Code-Smell-Code-Generation.

The contributions of this paper are: (i) a large-scale empirical study of code smell occurrence in the dataset and the output of transformer-based Python code generation techniques, (ii) an investigation of how transformer-based open-source techniques can differ from closed-source ones, and (iii) a discussion of the implications of the findings for researchers and practitioners.

II. Background

A. Code Smell

A **code smell** ("bad code smell" or "smell") is an indicator of an improper choice of system design and implementation strategy [15], [29]. These smells have been linked to signs of software maintainability issues. They also violate basic software design principles, which harm the product's future efficiency. These flaws can stifle software development or raise the chance of future errors or failures [30]. A code smell is a broader term that includes security issues, design decision issues, and coding standard violations. An example of a smell is *using the wrong exception catching order*, as shown in the code snippet below. The `TypeError` block is never reached, as the `Exception` block will catch all exceptions.

```
try:
    age = int(input())
except Exception:
    raise
except TypeError:
    raise
```

In our work, we distinguish code smells into ***security smells*** and ***non-security-related smells***. We make these distinctions because security smells can be more dangerous and problematic than non-security-related smells, as they can introduce vulnerabilities in code.

Security code smells (or simply "security smells") are a subset of code smells. They are code patterns frequently that may lead to security flaws [17], [18]. Although security smells may not be a vulnerability per se, they are symptoms that signal the prospect of a vulnerability [16]. In our scope, we are considering the security code smells that map to the Common Weakness Enumeration (**CWE**), which is a community-developed list of software and hardware security weakness [31]. For example, the following code snippet contains a security smell mapped to the *CWE-798: Use of Hard-coded Credentials*:

```
def verifyAdmin(password):
  if password != "passw0rd!":
    return False
  return True
```

This code snippet checks the password with respect to a hard-coded string (*i.e.*, `passw0rd!`). Although this code is functionally correct, anyone can bypass the authentication mechanism by guessing this fixed password. This can be achieved by, for example, reverse engineering the code to find the hard-coded password from the program's symbols table. Many instances of this security smell lead to authentication bypass in real software systems. One instance is the CVE-2010-2073 [32] in which the FTP server library (pyftpd/0.8.4.6) used hard-coded usernames and passwords for three default accounts (*"test"*, *"user"*, and *"roxon"*). It allowed remote attackers to read arbitrary files from the FTP server.

Non-security smells do not have a direct security implications. It is mainly related to potential errors such as using undefined variables and function redefining and standard coding practice violations. The standard coding practice encompasses a set of rules that developers need to follow for better code in terms of maintainability and readability. Different languages adopt specific coding practice protocols. For example, PEP-8 [33] is the *de facto* coding practice standard for Python. It provides an extensive guide for code layout, whitespace usage, naming conventions, *etc*.For example, according to the guide, the coding layout should have 4 spaces per indentation level, and spaces are the preferred indentation method.

B. Transformer-based Code Generation

Due to machine learning and natural language processing advancements, language learning models provide effectiveness in translation, question answering, and summarization. Similar techniques can be used for source code understanding and its related task. Source code generation from natural language is a Sequence-to-Sequence (seq2seq) learning problem. Previously, practitioners used the Recurrent Neural Network (RNN) to model the seq2seq problem, and with Long Short-Term Memory (LSTM) based neural network [34], it got significant improvement. Later, attention-based transformer [10] changed the area of language learning. The transformer is an encoder-decoder architecture-based deep learning model that uses the self-attention mechanism for differentially weighting the significance of each part of the input data [10]. The transformer architecture is what powers several popular language models such as BERT (Bidirectional Encoder Representations from Transformers) [35] and GPT-3 (Generative Pre-trained Transformer) [36]. These language learning models can be fine-tuned with source code-related datasets for code generation, understanding, and summarization. CodeBERT [37] and CodeT5 [38] are example of this type model. GitHub Copilot is a deep learning-based tool developed by GitHub and OpenAI [39] to help programmers write code, comments, test cases, documentation and translate source code from one programming language to another. The OpenAI Codex [2], an artificial intelligence model produced by OpenAI, powers GitHub Copilot. The OpenAI Codex is a modified, production-ready version of the GPT-3 model [36] that could generate text that resembles human language.

III. Research Questions

We aim to answer the following research questions:

RQ1: Are code smells present in the code generation training datasets?

We use two static analyzers (Bandit [40] and Pylint [41]) to detect smells in the samples of three training datasets used for Python code generation. We use Pylint to detect non-security code smells (*e.g.*, code convention violations) and Bandit to detect security smells that are mapped to a CWE ID.

RQ2: Does the output of an open-source transformer-based code generation technique contain code smells?

We evaluate the output from the different configurations of an open-source code generation model (GPT-Code-Clippy [27]) with Bandit and Pylint. We focus on having a qualitative result from the analyzers, and their correlation with learning harmful patterns related to code smells, especially security smells that have already been present in the training set.

RQ3: Is there any code smell in the output of closed source code generation tools based on a large language model?

In RQ1 and RQ2, we studied the training sets and the output of *open source* code generation techniques. This question focuses on the quality of the code generated by *closed source* tools (*i.e.*, systems in which their datasets, model, or trained weights are publicly unavailable). We aim to investigate whether proprietary systems may include quality/sanity checks in the output of their models and how they could differ from open-source ones. To answer this question, we analyze the code generated by GitHub Copilot. We use Bandit and Pylint to gather a quantitative result, running on the output from a commonly used dataset (HumanEval [2]).

IV. Methodology

Figure 1 shows an overview of the steps we performed in our study to answer each research question. We detail these steps in the following subsections.

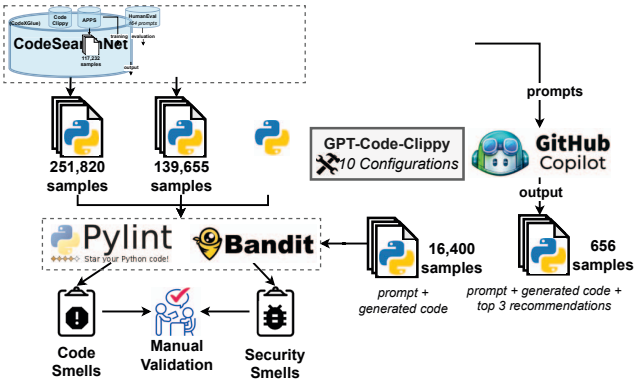

Fig. 1. Overview of our empirical study

A. RQ1: Code Smells in Training Datasets

In RQ1, we investigate the presence of code smells in the training sets of transformer-based code generation techniques. We used three datasets for our analysis: **CodeXGlue** [24] (which is a filtered version of the CodeSearchNet corpus [42]), the Automated Programming Progress Standard (**APPS**) [25], and **Code Clippy** [26]. We chose these datasets because they were used to train the GPT-Code-Clippy (GPT-CC) [27] model, which we use to answer our RQ2 (Section IV-B). Moreover, these datasets are used not only for code generation but also for model evaluation and code summarization [2], [24]. It is important to highlight that we use **CodeXGlue** [24] instead of CodeSearchNet corpus [42] because it contains code snippets that (i) could be successfully parsed into an abstract syntax tree, (ii) its corresponding natural language code document does not contain special tokens, (iii) have at least three and less than 256 tokens, and (iv) is written in English. That is, we chose CodeXGlue [24] because it is a noise-free version of the CodeSearchNet corpus [42].

To answer our RQ1, we first extracted Python code samples from these training datasets as follows:

- The APPS [25] training set contains 5,000 programming competition and interview problems with problem statements, sample inputs, and different solutions for each problem written in Python. We parsed each solution into a different Python file, obtaining a total of **117,232** samples.
- The CodeXGlue [24] contains around 6 million functions written in six different programming languages (Go, Java, JavaScript, PHP, Python, and Ruby). Their Python dataset is available in JSONL (JavaScript Object Notation Lines) format. Each line in the dataset is a JSON object containing a code snippet and a docstring with related metadata, such as the snippet's programming language, source repository, code tokens, and docstring. We parsed each JSON object and dumped them into individual Python files. In total, we have **251,820** Python files from this dataset.
- Code Clippy is a dataset created selecting GitHub repositories based on a set of criteria (*e.g.*, repository has at least 10 GitHub stars) [26] and then combined the GitHub portion from the Pile dataset [43]. After combining, duplicates were removed from the dataset. It includes samples from different programming languages. This dataset is provided in JSONL format. We used **139,655** Python samples from this dataset.

After collecting these Python code snippets, we run two static analyzers on them (Pylint [41] and Bandit [40]) with their default configuration. In the case of Pylint, we exclude *whitespaces*, *docstrings*, and *import-related* messages from our analysis (as explained in Section IV-D). Finally, we computed the following metrics:

- ***Number of smells per sample***: We calculated how many messages Pylint generated per sample.

- ***Top-5 message types***: We considered Pylint's messages in four categories: Error, Convention, Warning, and Refactor. We accumulated the weighted total number of instances found in each dataset for every type of code smell and ranked the top five in each category. This rank would help to understand the distribution of typical code smell present in the code generation dataset. The weighted total number of instances means the number of instances is divided by the total sample of the dataset.
- ***Number of "smelly" samples***: We calculated the number of samples that contained a code smell ("smelly sample"). This metric could help determine the percentage of the dataset responsible for the overall code smell in the dataset.
- ***Number of security smells per sample***: It indicates the number of security smells that Bandit could find in each sample. This metric is used to understand the quality of datasets based on the security smell.
- ***Top 3 security smells*** found for each dataset and their mapping to a CWE ID.

B. RQ2: Code Smells in the Generated Code

In this research question, we investigate whether there can be a leakage of code smells from training datasets into the generated code (*i.e.*, model's output). For this purpose, we gathered the code generated (*i.e.*, output) by the **GPT-Code-Clippy (GPT-CC)** [27]. The GPT-CC is a community attempt to produce an open-source version of GitHub Copilot.

The GPT-CC model was trained and fine-tuned using the CodeSearchNet [42], APPS [25] and Code Clippy [26] datasets. It was previously evaluated using the HumanEval dataset [2], which contains 164 *prompts* (*i.e.*, a partial code snippet containing a function's name, parameter list, and a comment describing the function's intended functionality). The GPT-CC provided ten outputs for each prompt (*i.e.*, 1,640 outputs per configuration). Since there are 10 different configurations, we have a total of **16,400** output samples.

After collecting the output samples, we combined the *original prompt* in the HumanEval dataset and the *model's output* and dumped them into individual Python files. Then, we run Pylint and Bandit on these files. Finally, we calculated the ***number of smells per sample*** from the Pylint result and the **number of security smell per sample** from the Bandit result. We also calculated the **top three messages** found in each category in the following way: for a specific type of message, we check the presence of the type in each configuration. For example, if Warning-A is present in nine configurations, whereas Warning-B is in eight configurations, then Warning-A has the higher rank. If there was a tie, we ranked the message with the accumulated number of instances in all configurations. In addition to that, we also calculated the percentage of samples containing code smell. For Bandit, the types of security smells are limited. Hence, we did not rank them; instead, we presented them all in our results (Section V-B).

C. RQ3: Code Smells in GitHub Copilot

We aim to understand the quality of a closed-source code generation model. For this purpose, we used GitHub Copilot [28] as an example of a closed-source code transformer-based code generation technique. We used their VS Code extension to obtain suggestions by giving the same 164 prompts from the HumanEval dataset we used in RQ2.

GitHub Copilot gave one suggestion in the text editor and generated ten additional suggestions. We collected the top three suggestions from the additional generated solutions for the 164 prompts. GitHub Copilot sometimes generates duplicate suggestions and hides them from the output. If the total additional suggestions are less than three, we treated the last suggestion as multiple suggestions. By following this procedure, we collected a total of **656 samples** from GitHub Copilot, which was later analyzed by Pylint [41] and Bandit [40] to check the presence of code smells.

Finally, we calculated the number of smells and security smells per sample. However, in this case, we did not rank the smell found by Pylint and Bandit, as the types of code smell found by analyzers are limited.

D. Code Smell Analyzers (Pylint and Bandit)

Pylint [41] is a static code analyzer for Python 2 and 3 that may display various messages in the following categories: ***fatal*** (when Pylint is unable to process the file), ***error*** (code smells that may lead to runtime errors), ***warning*** (Python-specific smells), ***convention*** (coding standard violations), and ***refactor*** (code smells that can be fixed through refactoring). We used Pylint 2.13.8, and we ignored these messages related to style issues about whitespace, newline, and invalid name: *C0303-trailing-whitespace*, *C0304-missing-final-newline*, *C0305-trailing-newlines*, and *C0103-invalid-name*. We also ignored these messages related to missing docstring as it is expected that samples in code generation datasets do not have a docstring: *C0112-empty-docstring*, *C0114-missing-module-docstring*, *C0115-missing-class-docstring*, and *C0116-missing-function-docstring*. Moreover, these `import` related messages are also ignored because Pylint can not do a reliable checking for `import` statements' usage [19]: *W0611-unused-import*, *W0401-wildcard-import*, *R0402-consider-using-from-import*, *C2403-non-ascii-module-import*, *W0404-reimported*, *W0614-unused-wildcard-import*, *C0410-multiple-imports*, *C0411-wrong-import-order*, *C0412-ungrouped-imports*, *C0413-wrong-import-position*, *C0414-useless-import-alias*, *C0415-import-outside-toplevel*, and *E0401-import-error*. We also omitted *fatal* error messages.

We used Bandit 1.7.4 [40] to statistically analyze samples for finding security smells. Bandit accomplishes this by processing each file, creating an Abstract Syntax Tree (AST), and applying suitable plugins to the AST nodes. Each detected security smell maps to a CWE ID.

E. Validation of the Analyzers

Static analyzers are known for including false positives in their results; for a dynamic language like Python, statically detecting smells can be difficult [44]. Therefore, we manually validated the output of Pylint and Bandit. These tools analyzed a total of **525,763** Python files, in which **508,707** of them are from the three training sets, **16,400** are from GPT-CC's output, and **656** samples are from GitHub Copilot's output.

Since it would be humanly unfeasible to manually verify over half a million Python files, to do our validation, we selected a statistically significant sample of **384** outputs from Pylint and **380** outputs from Bandit (confidence level = 95%, margin of error = 5%). The validated smells include issues detected in samples from the training sets and the open-source and closed-source models' output. We selected this subset to be proportional to the ratio of samples to the total population. Hence, we took 64 samples from APPS, 257 samples from CodeXGlue, 57 samples from Code Clippy, 5 samples from the output of GPT-CC, and 1 sample from the output of GitHub Copilot to validate Pylint. To validate Bandit, we took 25 samples from APPS, 166 samples from CodeXGlue, 186 samples from Code Clippy, 2 samples from GPT-CC's output, and 1 sample from GitHub Copilot's output.

After extracting these samples, we investigated whether the detected code smells were true positives. For Bandit, specifically, we checked the samples in terms of the possibility of exploitation of the security smell. The sample could be marked for different code smells, but in our validation, we focused on the smell for which it was detected by the tool.

We observed that Pylint correctly identified code issues for all the samples (***i.e.*, 100% precision**). Our manual investigation of Bandit showed that 345 out of 380 samples were correct (***i.e.*, 90.79% precision**). Therefore, this manual validation gave us confidence that our collected results are enough to allow an accurate analysis of code smells in transformer-based code generation techniques.

V. Results

A. RQ1: Code Smells in Training Sets

Table I presents the number of detected smell instances per type that were detected by Pylint, namely *error*, *convention*, *refactor* and *warning* smells. It also presents the total smell instances found per dataset, the number of samples containing smells (*i.e.*, "smelly samples"), and the average number of smell messages per sample. We observed that CodeXGlue was the dataset that contained the highest amount of "smelly" samples – Pylint detected smells in 97% of its samples. In the case of APPS and Code Clippy, the percentage of smelly samples was about 69% and 39%, respectively. Moreover, we found that the APPS [25] dataset has the lowest average number of smells per sample compared to the other datasets.

Table II shows the top five messages from each category found by Pylint. The last column represents the sum of the weighted total number of instances found in each dataset for every type of code smell, as explained in Section IV-A.

TABLE I
Result from Pylint on the Training Sets

Dataset	# Error Instances	# Convention Instances	# Refactor Instances	# Warning Instances	Total Smell Instances	# Smelly Samples	Avg. # Smells per Sample
APPS	61,294	74,341	106,478	278,041	520,154	81,068 (69.15%)	4.44
Code Clippy	273,852	66,976	103,114	76,212	3,386,765	54,641 (39.12%)	24.25
CodeXGlue	17,162	268,497	185,204	49,285	2,791,951	244,338 (97.03%)	11.09

TABLE II
Top-5 Code Smells Detected by Pylint on the Training Sets

Message Type	Message ID	Message	Weighted Score
Error	E0602	undefined-variable	9.813
	E0402	relative-beyond-top-level	0.189
	E1101	no-member	0.099
	E0611	no-name-in-module	0.029
	E1120	no-value-for-parameter	0.009
Convention	C0301	line-too-long	3.865
	C0209	consider-using-f-string	1.030
	C0321	multiple-statements	0.151
	C0200	consider-using-enumerate	0.097
	C0325	superfluous-parens	0.092
Refactor	R0903	too-few-public-methods	0.299
	R1705	no-else-return	0.255
	R0913	too-many-arguments	0.188
	R0914	too-many-locals	0.129
	R1725	super-with-arguments	0.102
Warning	W0311	bad-indentation	8.337
	W0104	pointless-statement	7.391
	W0212	protected-access	0.953
	W0613	unused-argument	0.339
	W0105	pointless-string-statement	0.328

TABLE III
Result from Bandit on Training Sets

Datasets	Syntax Error	# Security Smell Type	Total Smell Instances	# Security "Smelly" Samples	Avg. # Sec. Smell Per Samples
APPS	5,294 (4.51%)	15	2,533	1,903(1.62%)	0.0216
Code Clippy	74,192 (53.08%)	64	87,766	14,353(10.27%)	0.6284
CodeXGlue	2,123 (0.84%)	53	18,599	12,672(5.32%)	0.0739

Our results from Bandit show that the Code Clippy dataset [26] has the highest number of types and total security smells than other datasets. The number of security smell per sample is lower in APPS [25] than in the other two datasets. The top five security smells for these three datasets are B101: assert used in the code, B110: in try-except, using `pass`, without handling the exception, B311: blacklisted functions for random number generation, B603: subprocess_without_shell_equals_true, B307: blacklisted function for using `eval`. The first two are related to CWE-703 (Improper Check or Handling of Exceptional Conditions). The third one is related to CWE-330 (Use of Insufficiently Random Values). The last two maps to CWE-78 (Improper Neutralization of Special Elements used in an OS Command). Table III summarizes the total types of security smells found (2nd column), the number of security smells detected (3rd column), the number of samples containing security smells (4th column), and the average number of security smells per sample (5th column). The first column in Table III also includes the number of samples which Bandit

TABLE IV
TOP-3 SECURITY SMELLS DETECTED BY BANDIT ON THE TRAINING SET

Dataset	Message	CWE	Total
APPS	B307-blacklist(`eval`)	CWE-78: OS Command Injection	1,229 (48.52%)
	B110-try_except_pass	CWE-703: Improper Checking or Handling of Exceptional Conditions	591 (23.33%)
	B101-assert_used	CWE-703: Improper Handling of Exceptional Conditions	329 (12.99%)
Code Clippy	B101-assert_used	CWE-703: Improper Checking or Handling of Exceptional Conditions	66,247 (75.48%)
	B311-blacklist(`random`)	CWE-330: Use of Insufficiently Random Values	4,280 (4.88%)
	B603-subprocess_without_shell_equals_true	CWE-78: OS Command Injection	1,594 (1.81%)
CodeXGlue	B101-assert_used	CWE-703: Improper Checking or Handling of Exceptional Conditions	10,484 (56.37%)
	B603-subprocess_without_shell_equals_true	CWE-78: OS Command Injection	1,389 (7.47%)
	B110-try_except_pass	CWE-703: Improper Checking or Handling of Exceptional Conditions	1,233 (6.63%)

could not analyze due to a syntax error. It can be observed that about half of Code Clippy [26] can not be parsed due to syntax errors. Our observation is that though the sample files have a ".py" extension, they do not contain Python code; instead, they contain Markdown (.md) or Interactive Python Notebook (.ipynb) code.

Table IV presents the top three security smells found in each dataset and their mapping to a CWE entry. The last column of the table indicates the number of instances of this type of smell overall, and the percentage indicates the contribution of this smell to the total number of security smells in this particular dataset. For example, there are 2,533 security smell instances found in the APPS [25] dataset, in which 48.52% of them are instances of B307-blacklist(`eval`).

From Table I and III, we observe that 69.15% samples contributed to producing non-security code smells, whereas 1.62% samples contributed to security smell in APPS [25] dataset. Though the Code Clippy dataset [26] has the highest non-security and security smells per sample, we found that 39.12% and 10.27% of samples are contributing to code smells and security smells, respectively. In the CodeXGlue [24] dataset, almost every sample has non-security-related code smells identified by Pylint (97% smelly samples), whereas Bandit identified that 5.32% of its samples contain security smells. Among these training sets, APPS [25] has a lower non-security code smell and security smell per sample.

Comparison to prior study: A prior study [19] identified the top 10 code smells found in machine learning projects. Out of these 10, we considered 5 in our project (as explained in Section IV-D). *W0311: bad-indentation* is the top warning message, and *C0301: line-too-long* is the top convention smell found in our study and in [19]. *E1101: no-member* is the top error-related smell found in this prior study [19] and the third in our study. Though *R0801: duplicate-code* and *W0621: redefined-outer-name* are in the top 10 code smells in their study [19], we found that they are present in the code generation dataset but not as frequent.

RQ1 Findings: We found a total of 264 different types of non-security smells detected by Pylint for three commonly used datasets. *"Undefined variables"*, *"line too long"*, *"too few public methods"*, and *"bad indentation"* were the four most common non-security-related code smell identified across these datasets. We also found that a reoccurring security smell in the training sets is an *Improper Check or Handling of Exceptional Conditions* (CWE-703). Finally, the APPS [25] dataset was the one that had the lowest average number of smells per sample (both non-security related and security smells).

B. RQ2: Code Smells in Generated Code

We run Pylint and Bandit on the output of HumanEval [2] dataset from ten different configurations of the Fine-Tuned GPT-Neo 125M Parameters Model [27]. Table V provides the total type of non-security code smells found by Pylint. A configuration that has a name with "code-clippy" in it without "dedup" means that the model was fine-tuned with Code Clippy [26] dataset *with* duplication. A configuration name "code-clippy" with "dedup" means no duplication. The label "code-search-all" means that the model was fine-tuned with the full CodeSearchNet [42] dataset, whereas the label "code-search-py" indicates a fine-tuning with only the Python training set. The numbers `1024` and `2048` at the end of the configuration name indicate the *sequence size*, and `2048bs` means that the batch size is equals to 2048. One configuration was fine-tuned with APPS [25] dataset. The *base model* configuration means that the model was not fine-tuned with an external dataset like APPS [25].

Table VI shows the top-3 messages provided by Pylint after running it on the output of the ten different configurations of GPT-Neo 125M models based on the presence in the different configurations, as explained in Section IV-B. We found that *undefined variables* is a persistent smell in the output. Most model configurations generate *lines of more than 100 characters*, which is one of the typical code convention violation messages. The model generates *duplicated code*, and Pylint suggests refactoring it. An *unused argument* message is a common warning from Pylint for the model's output.

We run Bandit to find the security-related smell in the output from different configurations of fine-tuned GPT-Neo 125M model. We found that some models are not generating syntactically correct codes due to the model's performance. Table VII presents the type, the total number of security smells detected by Bandit, and the average number of security smells per sample. Bandit finds six security smell types in the output from

TABLE V
RESULT FROM PYLINT ON THE OUTPUT FROM DIFFERENT CONFIGURATIONS OF FINE-TUNED GPT-NEO 125M PARAMETERS

Configuration	# Error Instances	# Convention Instances	# Refactor Instances	# Warning Instances	Total Smell Instances	# Smelly Samples	Avg. # Smells per Sample
code-clippy-code-search-all	89	74	64	173	400	156 (9.51%)	0.244
code-clippy-code-search-py	136	73	61	166	436	159 (9.70%)	0.266
code-clippy-dedup-1024	237	109	60	489	895	319 (19.45%)	0.546
code-clippy-dedup-2048	184	21	0	225	430	71 (4.33%)	0.262
Base Model	288	108	83	356	835	292 (17.80%)	0.509
code-clippy-dedup-2048bs	281	132	81	444	938	314 (19.15%)	0.572
APPS	201	110	108	557	976	274 (16.70%)	0.595
code-search-all	1286	558	351	792	2987	1102 (67.20%)	1.821
code-clippy	57	150	6	635	848	469 (28.60%)	0.517
code-search-py	1322	561	492	696	3071	1132 (69.02%)	1.873

TABLE VI
TOP-3 MESSAGES FROM PYLINT ON THE OUTPUT FROM DIFFERENT CONFIGURATIONS OF FINE-TUNED GPT-NEO 125M PARAMETERS

Message Type	Message Id	Message	# of Presence
Error	E0602	Undefined-variable	10
	E1101	No-member	9
	E0601	Used-before-assignment	9
Convention	C0301	Line-too-long	10
	C0325	Superfluous-parenthesis	9
	C0200	Consider-using-enumerate	8
Refactor	R0801	Duplicate-code	9
	R1705	No-else-return	8
	R1710	Inconsistent-return-statements	8
Warning	W0613	Unused-argument	10
	W0104	Pointless-statement	10
	W0105	Pointless-string-statement	10

GPT-Neo: (i) B101-Assert used, (ii) B311-blacklist(`random`), (iii) B324-Hashlib, (iv) B112-try-except continue, (v) B307-blacklist(`eval`), and (vi) B110-try-except pass.

TABLE VII
RESULT FROM BANDIT ON THE OUTPUT FROM DIFFERENT CONFIGURATIONS OF FINE-TUNED GPT-NEO 125M PARAMETERS

Configuration	Syntax Error	# Security Smell Type	Total Smell Instance	# Security Smell Samples	# Security Smell Per Samples
code-clippy-code-search-all	660(40.24%)	1	10	3(0.18%)	0.006
code-clippy-code-search-py	656(40.00%)	1	10	5(0.30%)	0.006
code-clippy-dedup-1024	1286(78.41%)	1	11	6(0.37%)	0.007
code-clippy-dedup-2048	1568(95.61%)	1	5	3(0.18%)	0.003
Base Model	1314(82.13%)	1	9	6(0.37%)	0.005
code-clippy-dedup-2048bs	1286(78.41%)	2	21	11(0.67%)	0.013
APPS	1341(81,77%)	2	2	2(0.12%)	0.001
code-search-all	264(16.09%)	6	73	40(2.43%)	0.045
code-clippy	1167(71.16%)	0	0	0(0.00%)	0
code-search-py	260(15.85%)	4	43	27(1.64%)	0.026

RQ2 Findings: Code smells are present in the output of the fine-tuned GPT-Neo model's output. *Undefined variables*, *lines too long*, *Duplicate code*, and *Unused argument* are the top non-security smells in the training set and the model's output. Security smells, for example, *using assert*, are common in the generated suggestions.

C. RQ3: Code Smells in GitHub Copilot's Suggestions

We run Pylint [41], and Bandit [40] on the suggestions given in the IDE and on the top three suggestions from the ten suggestions generated by GitHub Copilot. Table VIII summarizes the total type of *error*, *convention*, *refactor*, and *warning* messages generated by Pylint. Table IX lists all non-security smell types detected by Pylint. The *undefined variable* is the only smell error found by Pylint in the GitHub Copilot's suggestions. We found 3 types of convention smells, 6 types of refactoring, and 7 types of warning smell from Pylint.

TABLE VIII
RESULT FROM PYLINT ON HUMANEVAL DATASET OUTPUT FROM GITHUB COPILOT

Suggestions	# Error Instance	# Convention Instances	# Refactor Instances	# Warning Instances	Total Smell Instances	# Smelly Samples	# Smells Per Sample
Top	12	57	18	22	109	76(46.31%)	0.664
First	9	55	29	23	116	84(51.22%)	0.707
Second	10	55	36	14	115	80(48.78%)	0.701
Third	8	51	31	11	101	75(45.73%)	0.616

TABLE IX
MESSAGES LIST FROM PYLINT ON HUMANEVAL DATASET OUTPUT FROM GITHUB COPILOT

Message Type	Message Id	Message
Error	E0602	Undefined-variable
Convention	C0200	consider-using-enumerate
	C0301	line-too-long
	C0123	unidiomatic-typecheck
Refactor	R1710	inconsistent-return-statements
	R1705	no-else-return
	R1719	simplifiable-if-expression
	R1703	simplifiable-if-statement
	R1716	chained-comparison
	R1718	consider-using-set-comprehension
Warning	W0612	unused-variable
	W0108	unnecessary-lambda
	W0107	unnecessary-pass
	W0127	self-assigning-variable
	W0311	bad-indentation
	W0105	pointless-string-statement
	W0622	redefined-builtin

We found two types of security smells in the outputs generated by GitHub Copilot: B101-Assert Used and B303-Blacklist (Use of insecure MD2, MD4, MD5, or SHA1 hash function). Table X presents the security smells detected in GitHub Copilot's generated code.

RQ3 Findings: GitHub Copilot provides executable suggestions, but they contain substandard coding and security smells. Undefined variables, long lines, inconsistent return statements, and unused variables are common code smells in different categories. GitHub Copilot's suggestions for the

TABLE X
RESULT FROM BANDIT ON HUMANEVAL DATASET OUTPUT FROM GITHUB COPILOT

Suggestions	Syntax Error	# Security Smell Type	Total Smell Instance	# Security Smell Samples	# Security Smell Per Samples
Top	3(1.83%)	2	3	2(1.22%)	0.018
First	3(1.83%)	1	1	1(0.61%)	0.006
Second	6(3.66%)	1	1	1(0.61%)	0.006
Third	5(3.05%)	2	2	2(1.22%)	0.012

HumanEval dataset contain security smells, such as *using assert* and *weak hash functions*.

VI. DISCUSSION AND IMPLICATIONS

This section first discusses the common non-security, and security code smells found in training sets and output for code generation tasks by Pylint. Subsequently, we discuss the implication of our findings.

A. Common (Non-Security) Code Smells

- **Undefined variables:** Most datasets are taken from open-source projects and may be part of complex projects. Taking a function or class from an arbitrary project location may miss important context; they may depend on the other part of the code base or external library. For example, the following partial code sample is taken from the CodeXGlue [24] Python training set with slight modification where `pexpect` on line 11 is not defined within the context. This smell is also found in the synthesized output of the GPT-Code-Clippy [27] model and GitHub Copilot [28].

```
1  def disconnect(self, driver):
2      """Disconnect from the console."""
3      self.log("TELNETCONSOLE disconnect")
4      try:
5          while self.device.mode != 'global':
6              self.device.send('exit',\
7                  timeout=10)
8      except OSError:
9          self.log("TELNETCONSOLE already\
10             disconnected")
11     except pexpect.TIMEOUT:
12         self.log("TELNETCONSOLE unable to\
13             get the root prompt")
14     ...
```

- **Not using enumerate**: Instead of using `enumerate`, developers use the `range` function, passing to it the result of the `len` function from built-in Python data structures. Using `enumerate` is very helpful to get the index and value at the same time and good practice to use in the code instead of accessing `len` and getting the value with index received by iterating with `range`. The following example is taken from the output of GitHub Copilot on a sample from the HumanEval dataset [2]. In lines 8 and 9, `enumerate` could be used and `number[i]` and `number[j]` at line number 10 could be available earlier if `enumerate` is used.

```
1  from typing import List
2  def has_close_elements(numbers: List[float],
3      threshold: float) -> bool:
4      """ Check if in given list of numbers,
5      are any two numbers closer to each other
6      than given threshold.
7      """
8      for i in range(len(numbers)):
9          for j in range(i + 1, len(numbers)):
10             if abs(numbers[i] - numbers[j])\
11                     < threshold:
12                 return True
13     return False
```

- **Inconsistent return statements**: If a return statement returns an expression, all return statements that return no value should explicitly return `None`, and an explicit return statement should be present at the end of the function, according to PEP8 [33] (if reachable). This refactoring message is found in the training set, the output of the fine-tuned model, and GitHub Copilot's synthesized output. The following function is taken from the APPS dataset [25]. At the end of the function, there should be a return statement in case the `if` condition at line number 13 is false. This function will implicitly return `None`, which may not be an expected output for this function. This problem is present in the training dataset and output of the fine-tuned model and GitHub Copilot.

```
1  def solution(string, ending):
2      if ending == "":
3          return True
4      sum = 0
5      if len(string) < len(ending):
6          return False
7
8      for i in range(1,len(ending)+1):
9          if string[-i] == ending[-i]:
10             sum += 1
11         else:
12             return False
13     if sum == len(ending):
14         return True
```

B. Frequent Security Code Smells

From Section V, we observe that security code smells are frequent in the training set and output of the code generation model. In this section, we discuss three frequently occurred security code smells identified by Bandit:

- **Using Assert**: Using `assert` in the production code is a bad practice; it can be replaced with proper `try-except` handling. This security smell is related to CWE-703 (Improper Check or Handling of Exceptional Conditions). This partial code has been taken from the APPS dataset [25] where at line 12, the `assert` is used to enforce the value of the variable `diff` is greater than or equal to 0.

```
1  class Solution:
2      def canConvertString(self, s: str, t: str,
3                           k: int) -> bool:
4          if len(s) != len(t):
5              return False
6          shifts = []
7          for ch1, ch2 in zip(s, t):
8              asc1 = ord(ch1) - ord('a')
9              asc2 = ord(ch2) - ord('a')
10             diff = asc2 - asc1 if asc2 >= asc1\
11                 else 26 - asc1 + asc2
12             assert diff >= 0
13             if diff > 0:
14                 shifts.append(diff)
```

- **Blacklisted Function**: Bandit checks the usage of Python function calls that have possible security implications. Dif-

ferent kinds of blacklisted functions can be found in the training sets. For example:

1) `eval`: This Python function is used to evaluate arbitrary Python expressions from a text or compiled code input. It can dynamically evaluate Python expressions from any input, whether a string or a built code object. However, the problem is that it is considered insecure as it could execute arbitrary Python code. The safer option for this function could be `ast.literal_eval`. This security smell is associated with CWE-78 (OS Command Injection).

2) `random`: Standard pseudo-random generators in Python could be handy for generating random numbers, but they are not suitable for cryptography. A cryptographically secure pseudo-random number generator is a random number generator that uses synchronization mechanisms to ensure that no two processes generate the same random number at the same time [45]. Python's `random` function is not suitable for this purpose. This security smell maps to CWE-330 (Use of Insufficiently Random Values).

This partial and slightly modified code is taken from the Code Clippy dataset [26]. In line 4, it uses invokes the `random.randint` function, which is a blacklisted function.

```
1 ...
2 # perform random check to assert
3 # the probability is valid
4 checkid = random.randint(0,len(resp_length)-1)
5 if resp_length[checkid] < 2:
6 ...
```

- **Subprocess without Shell Equals True**: The `subprocess` module in Python is used to run a new application by creating a new process from Python. When the `shell` parameter is equals to `true`, the code is executed through the system's shell (*e.g.*, /bin/sh). It is a good practice to invoke a subprocess *without* using a shell to prevent shell injection attacks. However, any user-provided or variable input without sanitizing has security implications, even if the shell argument is equal to false. Hence, Bandit [40] marks a subprocess invocation without a shell as a smell, as every user-provided input should be properly validated (but it has lower severity as compared to invoking it with `shell=true`). This security smell is related to CWE-78 (OS Command Injection). The following (slightly modified) code snippet is collected from the CodeXGlue dataset [24]. Line 7 has a call for opening a subprocess without the `shell` parameter equal to true but with a potential unsanitized input (`project_name`).

```
1 def runserver(project_name):
2         '''
3         Runs a python CGI server in a subprocess.
4         '''
5         ...
6         os.chdir(CGI_FOLDER)
7         subprocess.Popen("python -m\
8          http.server --cgi 8000 --name " + project_name)
```

C. Implication of the Findings

Bad code patterns can (and will) leak to the output of models: In RQ1, we observed that code smells occur in the training datasets. One of the training sets had 97% of samples with smells in them. When we observed the output of two code generation tools (GPT-Code-Clippy [27] and Github Copilot [28]), we found that these smells leaked to the output of these models. For example, in a configuration, GPT-Neo [46] model is fine-tuned only with the APPS [25] dataset and in the output for HumanEval dataset [2] from that configuration, we observed that the types of code and security smell are a subset of the types of code and security smell found in the APPS [25] training set.

Code generated by tools should be taken with a "grain of salt": Automated code generation has become very popular, especially after the release of GitHub Copilot in the last week of June 2021 as an extension in different text editors and IDEs. While code generation tools can help developers write runnable code from a docstring and previous code context, our study shows that the output of these models is not free from code smells and, more dangerously, security code smells. Using the generated code "as is" without properly checking its quality can introduce quality problems that may persist throughout the software development. Developers can consider using static analyzers and/or linters when accepting code generation suggestions made by these techniques.

Preprocessing of datasets: The training sets for code generation suffer quality problems. Hence, more research is needed to investigate and develop preprocessing techniques to remove and/or repair such quality problems. Automated tools can quickly fix some smell types, and our study on security smell in training set by Bandit shows that less than 11% of the training samples contain all the security smell. Removing them from the training set could be a way to improve the output quality of the model.

VII. Threats to Validity

The main threat to the construct validity of our work is that our analysis heavily depends on the accuracy of the tools we used (*i.e.*, Pylint and Bandit). To mitigate this threat, we employed a *systematic process* to manually validate random samples such that we could determine the confidence level of our results (as described in Section IV-E). Another threat relates to the generalizability of the findings of the work (*external validity*). We inspected smells across three open-source datasets, which are written in Python. Thus, the results may not generalize to code generation techniques for other programming languages. Furthermore, though the datasets are machine learning model-independent, we only focused on the output of the fine-tuned GPT-Neo [46] model and GitHub Copilot [28]. Both of them are GPT-style [36] models. There are other transformer-based models for generative task, *i.e.*, CodeBERT [37] and CodeT5 [38]. Our results may not be generalized for these models. However, it is important to highlight that this work did not aim for *statistical generalization* but rather *analytical*

generalization; the three datasets were carefully chosen from various sources. Hence, we anticipate that these datasets will reflect a typical dataset for training transformer-based code generation techniques. For the generative models, we only focused on the output to understand the presence of smells and checked ten configurations of an open source model along with a closed source model. A more extensive study can be conducted to create a correlation between code smell and the model's hyperparameters.

Finally, we only consider security smells that map to the CWE list. We acknowledge that it may not include all possible security smell types. However, it is important to highlight that the CWE list is a mature community-established list of security weaknesses that have been observed and documented in the real world and have been widely used by the security community (both in academia and industry).

VIII. Related Work

Prior works have focused on developing an automated generation of source code that can implement a given task. Most of these techniques take as input the task described in natural language with a set of input and output examples. Before the emerging techniques of the deep learning process, program synthesis was formulated as searching for the program within a defined search space. Gulwani et al. [47] surveyed the foundation of program synthesis, such as the deductive synthesis approach [48], [49], in which task specification is converted into constraints and extracts the program after proving the constraint satisfaction. Yin *et al.* [50] employed recurrent networks to map text to abstract syntax trees and then code using attention. Large language learning models have been excellent in generating source code from the given context. After fine-tuned on large code dataset, a variety of large language learning models have been released aiming to generated code (*e.g.*, CodeBert [37], Codex [2], CodeT5 [38]). AlphaCode [51] is another type of code generation model that generates code for competitive programming problems. Codex's updated version is later used to create an advanced auto-complete system, GitHub Copilot [28]. Our work focuses on the code smells in the training sets and their leakage to the model's generated output instead of evaluating the performance of transformer-based [10] fine-tuned code generation machine learning models.

Code smells are ongoing quality issues in a project's maintainability, and readability [15]. Manual detection of the smell is time-consuming and may not scale [52]. Hence, several works focused on developing techniques to detect code smells automatically [53]–[55]. Lanza and Marinescu [52] proposed a metric-based detection strategy to detect code smells. Chen *et al.* [56] implemented a code smell detection tool, Pysmell, which can detect 11 code smells in the Python project. Di Nucci *et al.* [57] described machine learning-based experiments with a new dataset configuration that includes instances of multiple types of smells. Though the authors decided that using the machine learning model still needs much work, they proposed that it can be improved.

Other works studied smells related to coding standard violations. The coding style guide must be followed as it is directly related to readability. It enforces to follow a certain style from the naming convention of the variable to the coding layout [58]. Without a good readable code, it could hamper the maintainability [59]. To enforce following the coding standard, automated tools are available that could validate code quality [60]. For example, Simmons *et al.* [20] used Pylint [41] to analyze the coding standard of 1,048 open-source data science projects. In our work, we used existing detectors to analyze the persistence of code smells; specifically, security smells in datasets and the output of transformer-based source code generations.

With the rise of code generation tools integrated with IDEs, prior works studied to what extent the generated code is *functionally correct* (*i.e.*, it implements what the user expected) [2], [11], [21]. However, these works did not verify the *quality* of the generated code. More recent works investigated whether GitHub Copilot can generate vulnerable code [22] or whether it is as bad as humans in generating insecure code [23]. Although these works point to potential quality issues in automatically generated code, we currently lack an in-depth understanding of the quality of training datasets and whether code smells can potentially leak into the code generated by these models. Therefore, our work conducts a systematic large-scale empirical study on the training sets and outputs of two transformer-based code generation techniques.

IX. Conclusion

Automated code generation can help developers reduce the time spent writing code with common patterns. In this paper, we investigated whether the code generated by transformer-based code generation techniques can introduce code smells. By performing a large-scale empirical study over training sets, we found that they contain several smell occurrences, where undefined variables were the most common ones, and the use of dangerous functions were the most reoccurring smells. We also investigated whether the smells from the training set can leak to the generated code models. We found that both an open-source (GPT-CC) and a closed-source (GitHub Copilot) code generation tool generated problematic code. This study highlights the importance of not accepting generated code as-is and provides a motivation for further research in techniques that can generate code that is not only functionally correct but also would not introduce quality issues.

References

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto *et al.*, "Evaluating large language models trained on code," 2021.

[3] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," in *44th International Conference on Software Engineering (ICSE)*, 2022.

[4] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 150–162.

[5] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. V. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 329–340.

[6] Y. Gao and C. Lyu, "M2ts: Multi-scale multi-modal approach based on transformer for source code summarization," *arXiv preprint arXiv:2203.09707*, 2022.

[7] A. V. M. Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," *arXiv preprint arXiv:1707.02275*, 2017.

[8] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 05, 2020, pp. 8984–8991.

[9] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[11] S. Tipirneni, M. Zhu, and C. K. Reddy, "Structcoder: Structure-aware transformer for code generation," 2022. [Online]. Available: https://arxiv.org/abs/2206.05239

[12] T. Sharma, M. Fragkoulis, and D. Spinellis, "House of cards: code smells in open-source c# repositories," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 424–429.

[13] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

[14] V. Gudivada, A. Apon, and J. Ding, "Data quality considerations for big data and machine learning: Going beyond data cleaning and transformations," *International Journal on Advances in Software*, vol. 10, pp. 1–20, 07 2017.

[15] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[16] M. Ghafari, P. Gadient, and O. Nierstrasz, "Security smells in android," in *2017 IEEE 17th international working conference on source code analysis and manipulation (SCAM)*. IEEE, 2017, pp. 121–130.

[17] M. R. Rahman, A. Rahman, and L. Williams, "Share, but be aware: Security smells in python gists," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 536–540.

[18] A. Rahman, C. Parnin, and L. Williams, "The Seven Sins: Security Smells in Infrastructure as Code Scripts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 164–175. [Online]. Available: https://ieeexplore.ieee.org/document/8812041/

[19] B. van Oort, L. Cruz, M. Aniche, and A. van Deursen, "The prevalence of code smells in machine learning projects," in *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*. IEEE, 2021, pp. 1–8.

[20] A. J. Simmons, S. Barnett, J. Rivera-Villicana, A. Bajaj, and R. Vasa, "A large-scale comparative analysis of coding standard conformance in open-source data science projects," in *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, oct 2020. [Online]. Available: https://doi.org/10.1145%2F3382494.3410680

[21] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *ArXiv*, vol. abs/2108.07732, 2021.

[22] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 980–994. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00057

[23] O. Asare, M. Nagappan, and N. Asokan, "Is github's copilot as bad as humans at introducing vulnerabilities in code?" *arXiv preprint arXiv:2204.04741*, 2022.

[24] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[25] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring coding challenge competence with apps," 2021. [Online]. Available: https://arxiv.org/abs/2105.09938

[26] N. Coooper, A. Arutiunian, S. Hincapié-Potes, B. Trevett, A. Raja, E. Hossami, M. Mathur *et al.*, "Code Clippy Data: A large dataset of code data from Github for research into code language models," Oct. 2021. [Online]. Available: https://github.com/CodedotAl/gpt-code-clippy/wiki/Dataset

[27] ——, "GPT Code Clippy: The Open Source version of GitHub Copilot," Jul. 2021. [Online]. Available: https://github.com/CodedotAl/gpt-code-clippy/wiki

[28] Github copilot : Your ai pair programmer. [Online]. Available: https://copilot.github.com

[29] M. Fowler, "CodeSmell." [Online]. Available: https://martinfowler.com/bliki/CodeSmell.html

[30] J. Pereira dos Reis, F. Brito e Abreu, G. de Figueiredo Carneiro, and C. Anslow, "Code Smells Detection and Visualization: A Systematic Literature Review," *Archives of Computational Methods in Engineering*, vol. 29, no. 1, pp. 47–94, Jan. 2022. [Online]. Available: https://doi.org/10.1007/s11831-021-09566-x

[31] The MITRE Corporation, "CWE - Common Weakness Enumeration," 2022. [Online]. Available: http://cwe.mitre.org/

[32] "CVE-2010-2073." [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2073

[33] N. C. Guido van Rossum, Barry Warsaw, "Pep 8 — the style guide for python code." [Online]. Available: https://pep8.org/

[34] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf

[35] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2018. [Online]. Available: https://arxiv.org/abs/1810.04805

[36] T. B. Brown, B. Mann, N. Ryder *et al.*, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[37] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[38] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," 2021. [Online]. Available: https://arxiv.org/abs/2109.00859

[39] OpenAI, "OpenAI," Jun. 2021. [Online]. Available: https://openai.com

[40] Bandit. [Online]. Available: https://bandit.readthedocs.io/

[41] Pylint. [Online]. Available: https://pylint.pycqa.org/

[42] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," 2019. [Online]. Available: https://arxiv.org/abs/1909.09436

[43] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima *et al.*, "The pile: An 800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020.

[44] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, and B. Xu, "Understanding metric-based detectable smells in python software: A comparative study," *Information and Software Technology*, vol. 94, pp. 14–29, 2018.

[45] C. Dufour, "How to ensure entropy and proper random numbers generation in virtual machines." [Online]. Available: https://www.exoscale.com/syslog/random-numbers-generation-in-virtual-machines/

[46] S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman, "GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5297715

[47] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[48] C. Green, "Application of theorem proving to problem solving," in *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, ser. IJCAI'69. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1969, p. 219–239.

[49] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Commun. ACM*, vol. 14, no. 3, p. 151–165, mar 1971. [Online]. Available: https://doi.org/10.1145/362566.362568

[50] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," 2017. [Online]. Available: https://arxiv.org/abs/1704.01696

[51] Y. Li, D. H. Choi, J. Chung, N. Kushman, J. Schrittwieser *et al.*, "Competition-level code generation with alphacode," *ArXiv*, vol. abs/2203.07814, 2022.

[52] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[53] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[54] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *Journal of Software Engineering Research and Development*, vol. 5, no. 1, pp. 1–28, 2017.

[55] F. L. Caram, B. R. D. O. Rodrigues, A. S. Campanelli, and F. S. Parreiras, "Machine learning techniques for code smells detection: a systematic mapping study," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 02, pp. 285–316, 2019.

[56] Z. Chen, L. Chen, W. Ma, and B. Xu, "Detecting code smells in python programs," in *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, 2016, pp. 18–23.

[57] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 612–621.

[58] T. Lee, J. B. Lee, and H. P. In, "A study of different coding styles affecting code readability," 2013.

[59] M. Elish and J. Offutt, "The adherence of open source java programmers to standard coding practices," 01 2002.

[60] S. Dasgupta and S. Hooshangi, "Code quality: Examining the efficacy of automated tools," 2017.