SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques

Mohammed Latif Siddiq
msiddiq3@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Joanna C. S. Santos
joannacss@nd.edu
University of Notre Dame
Notre Dame, IN, USA

ABSTRACT

Automated source code generation is currently a popular machine-learning-based task. It can be helpful for software developers to write functionally correct code from a given context. However, just like human developers, a code generation model can produce vulnerable code, which the developers can mistakenly use. For this reason, evaluating the security of a code generation model is a must. In this paper, we describe SecurityEval, an evaluation dataset to fulfill this purpose. It contains 130 samples for 75 vulnerability types, which are mapped to the Common Weakness Enumeration (CWE). We also demonstrate using our dataset to evaluate one open-source (*i.e.*, InCoder) and one closed-source code generation model (*i.e.*, GitHub Copilot).

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software development techniques*.

KEYWORDS

dataset, common weakness enumeration, code generation, security

ACM Reference Format:
Mohammed Latif Siddiq and Joanna C. S. Santos. 2022. SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques. In *Proceedings of The 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S 2022).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Code generation techniques are used to generate functional source code from a given *prompt*, which could be a comment, an expression in the form of the function signature, or their mixture [2]. By using these tools, developers can save time and reduce software development efforts and costs. Recently, machine learning-based techniques have been heavily used in source code generation tools. Large Language Learning Models (LLM) using attention-based transformer technique [30] are pre-trained with textual data, including source code snippets. Later, they are fine-tuned for specialized source code related tasks such as automated code summarization [10], completion [14, 15, 29], generation [27, 28] and documentation creation [4].

Although machine learning-based code generation techniques can generate functionally correct code, they may not be free from code smells or software vulnerabilities [20, 26]. Since they are trained on open-source projects, which may contain security flaws [12, 24, 25], these machine learning models can capture those flaws and leak them to the model's output. Hence, it is crucial to validate the output of such learning-based code generation techniques so that the generated code is not only *functionally* correct, but it also does not introduce a *vulnerability / insecure coding practice*.

In this paper, we present **SecurityEval**, a manually curated dataset for evaluating machine-learning-based code generation models from the perspective of software security. We collected Python samples of different vulnerability types, covering multiple categories from the Common Weakness Enumeration (CWE) [17]. Our dataset contains **130** samples representing **75** distinct vulnerability types (CWEs). These samples are formatted as *prompts* that could be used for a generalized source-code generation model. We released this dataset in our repository: https://github.com/s2e-lab/SecurityEval.

2 DATASET CONSTRUCTION

We created an evaluation dataset to measure the code quality generated by a machine learning model from the perspective of secure coding practices. We focused on collecting samples for the Python programming language because it is currently the most popular language [5] and is a language developers want to work with the most [1]. The following sections describe the sample collection steps and how these samples were formatted to meet our goal.

2.1 Samples Collection

We mined software vulnerability examples with their mapping to a CWE entry from four external sources:

- **CodeQL** [11] is a semantic code analysis engine from GitHub that can be used to query code and detect vulnerabilities. Its documentation includes different examples of source code with bad and good patterns. Hence, we inspected its documentation and retrieved a total of **36** Python samples containing bad patterns.
- The **Common Weakness Enumeration (CWE)** [17] is a well-known resource for researchers and practitioners. It enumerates common software and hardware weaknesses that lead to a vulnerability. Almost every entry in the CWE list provides examples of insecure code in different programming languages (*e.g.*, Java, C, PHP *etc.*). We extracted a total of **11** Python samples from it.
- **Sonar Rules**: *SonarSource* [23] is a company that has a static analyzer for finding code problems in multiple programming

MSR4P&S 2022, 18 November, 2022, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

languages. Its static analyzer contains around 4,800 rules to find implementation issues, such as bugs, vulnerabilities, security hotspots, and code smells. For Python, they have a total of 217 rules, including 29 vulnerability-related rules. The online documentation of these rules contains compliant and non-compliant examples. Thus, we retrieved **34** samples of non-compliant examples from it.

- **Pearce *et al.* [20]** investigated the frequency and circumstances in which GitHub Copilot may generate insecure code. The study focused on 18 CWEs to create different scenarios for GitHub Copilot, where most of the scenarios are adapted from CodeQL [11] and for different languages. We included **4** of their Python examples in our dataset.

We chose the first three sources because they are resources widely used by researchers and practitioners when studying vulnerabilities. Furthermore, we included samples by Pearce *et al.* [20] because, to our knowledge, it is the first peer-reviewed work to investigate security problems in ML-based code generation techniques.

After collecting the samples above, we obtained a total of 85 samples. Therefore, to further enrich our dataset, we created extra **45** examples ourselves. Though almost every entry in the CWE list has examples in different programming languages, they are mainly written in Java, C/C++, PHP, C#, and Perl. Since these weaknesses can be present in other programming languages besides the ones exemplified in the CWE entry, we follow the same pattern/structure described in the provided examples to create an example of insecure code in Python. We focused on covering vulnerability types (CWEs) other than the ones already covered by the 85 samples previously collected.

2.2 Samples Formatting

For ML-based code generation techniques, we need to provide the model with a ***prompt*** that will provide some context. With the prompt given as input, the code generation model generates the code. Therefore, after collecting the sample code snippets, we manually converted them to a specific format suitable for code generation techniques.

Our dataset is released following JavaScript Object Notation Lines (JSONL) format, in which each line contains a JSON object. The JSON object has three key-value pairs:

- **ID:** It identifies the sample uniquely. The ID is formatted in this way: `{CWE-ID}_{Source}_{Serial}.py`. The *CWE-ID* indicates which CWE this sample is related to. *Source* indicates from where the sample was extracted (*i.e.*, CodeQL [11], the CWE list [17], Sonar Rules [23], Pearce *et al.* [20] or created by the authors). *Serial* is a numerical value starting from 1. For example, an ID equals to `CWE-020_codeql_1` means this sample is for evaluating CWE-020: Improper Input Validation, is adapted from CodeQL [11] example, and it is first in serial.
- **Prompt**: The prompt is a partial source code that can be used as input for code generation techniques. It typically contains code statements, such as a function definition together with arguments and a docstring that describes the method's functionality, as well as `import` statements, variable declarations, *etc.*
- **Insecure Code**: It provides an example of possible vulnerable code that can be generated by models. The insecure code outlines how the prompt can lead to vulnerable code. These insecure codes may contain other vulnerabilities and may not fully cover the intended function of the prompt.

For instance, the code snippet below[1] has an example of a sample created by the authors for the CWE-798 (*Use of Hard-coded Credentials*). The prompt is a function (`getDBConnection`) and a *docstring* about the expected function behavior (*i.e.*, to *"return a mysql database connection"*). The "insecure code" attribute shows an example of a code snippet that hardcodes the expected username and password when connecting to a MySQL database.

```
{"ID": "CWE-798_author_1.py",
 "Prompt": "import mysql.connector
    def getDBConnection():
        '''
            return a mysql database connection
        ''''",
 "Insecure_code": "import mysql.connector
    def getDBConnection():
        '''
            return a mysql database connection
        '''
        return mysql.connector.connect(user='user', password='password',
                    host='host', database='database')"}
```

2.3 Dataset Overview

As shown in Table 1, our SecurityEval dataset has a total of **130** samples, covering **75** vulnerability types (CWEs). The first column in this table indicates the vulnerability type (CWE), and the four remaining columns are for the respective data source with the number of examples taken from them. The sixth column presents the number of examples for a particular CWE collected.

According to the CWE list version 4.8 [17], weaknesses related to *software development* are categorized into 40 categories. We cover 28 categories out of these 40 categories. We exclude the following categories as they are not related to Python or do not have enough explanation from the context of Python: *Complexity Issues*, *Documentation Issues*, *Encapsulation Issues*, *Memory Buffer Errors*, *Pointer Issues*, *String Errors*, *Lockout Mechanism Errors*, *Permission Issues*, *Signal Errors*, *State Issues*, *Type Errors*, and *User Interface Security Issues.*

3 APPLICATION

Our dataset can be used to investigate the security of code generation techniques by giving our prompts to the technique and then inspecting the generated code. This inspection can be performed ***manually*** or ***automatically***. For example, one can manually compare each generated code to the insecure code samples in our dataset. Alternatively, a researcher can rely on existing static analyzers (*e.g.*, Bandit) to automatically find vulnerabilities in the generated code and then rely on the alarms raised by the tool. If the alarm raised by the tool matches the CWE associated with the prompt, the generated code is likely insecure.

In the next section, we walk through an example of using the SecurityEval dataset to evaluate the security of code generated by a closed-source (*i.e.*, GitHub Copilot) and an open-source (*i.e.*, InCoder) code generation tool. These two models are chosen only

[1] We added indentation to this snippet for clarity. In the actual JSONL file in the released dataset, all JSON objects are flattened out in a single line.

Vulnerability Type (CWE)	Code QL	CWE List	Sonar Rules	Pearce *et al.*	Authors	Total
CWE-020 Improper Input Validation	4	0	0	0	2	6
CWE-611 Improper Restriction of XML External Entity Reference	1	0	4	0	1	6
CWE-601 Open Redirect	1	0	4	0	0	5
CWE-022 Path Traversal	2	0	0	0	2	4
CWE-297 Improper Validation of Certificate with Host Mismatch	0	0	4	0	0	4
CWE-327 Use of a Broken or Risky Cryptographic Algorithm	4	0	0	0	0	4
CWE-502 Deserialization of Untrusted Data	1	1	1	0	1	4
CWE-079 Cross-site Scripting	2	0	1	0	0	3
CWE-094 Code Injection	1	0	1	0	1	3
CWE-117 Improper Output Neutralization for Logs	1	0	1	0	1	3
CWE-295 Improper Certificate Validation	1	0	0	0	2	3
CWE-347 Improper Verification of Cryptographic Signature	0	0	3	0	0	3
CWE-703 Improper Check or Handling of Exceptional Conditions	0	0	0	0	3	3
CWE-730 Regex Injection	2	0	0	0	1	3
CWE-078 OS Injection	1	0	0	0	1	2
CWE-089 SQL Injection	1	0	0	0	1	2
CWE-090 LDAP Injection	2	0	0	0	0	2
CWE-113 HTTP Response Splitting	0	0	2	0	0	2
CWE-116 Improper Encoding or Escaping of Output	1	0	0	0	1	2
CWE-215 Insertion of Sensitive Info. Into Debugging Code	1	0	0	0	1	2
CWE-259 Use of Hard-coded Password	0	0	0	0	2	2
CWE-319 Cleartext Transmission of Sensitive Information	0	0	0	0	2	2
CWE-326 Inadequate Encryption Strength	0	0	0	0	2	2
CWE-434 Unrestricted Upload of File with Dangerous Type	0	0	0	2	0	2
CWE-521 Weak Password Requirements	0	0	2	0	0	2
CWE-522 Insufficiently Protected Credentials	0	0	0	1	1	2
CWE-643 XPath Injection	1	0	1	0	0	2
CWE-798 Use of Hard-coded Credentials	1	0	0	0	1	2
CWE-918 Server-Side Request Forgery (SSRF)	2	0	0	0	0	2
CWE-080 Basic XSS	0	0	0	0	1	1
CWE-095 Eval Injection	0	0	0	0	1	1
CWE-099 Resource Injection	0	0	1	0	0	1
CWE-1204 Generation of Weak Initialization Vector (IV)	0	0	1	0	0	1
CWE-193 Off-by-one Error	0	0	0	0	1	1
CWE-200 Exposure of Sensitive Info. to an Unauthorized Actor	0	0	0	0	1	1
CWE-209 Generation of Error Msg. Containing Sensitive Info.	1	0	0	0	0	1
CWE-250 Execution with Unnecessary Privileges	0	1	0	0	0	1
CWE-252 Unchecked Return Value	0	0	0	0	1	1
CWE-269 Improper Privilege Management	0	1	0	0	0	1
CWE-283 Unverified Ownership	0	1	0	0	0	1
CWE-284 Improper Access Control	0	0	0	0	1	1
CWE-285 Improper Authorization	1	0	0	0	0	1
CWE-306 Missing Authentication for Critical Function	0	0	0	1	0	1
CWE-312 Cleartext Storage of Sensitive Information	1	0	0	0	0	1
CWE-321 Use of Hard-coded Cryptographic Key	0	0	0	0	1	1
CWE-329 Generation of Predictable IV with CBC Mode	0	0	1	0	0	1
CWE-330 Use of Insufficiently Random Values	0	0	0	0	1	1
CWE-331 Insufficient Entropy	0	0	0	0	1	1
CWE-339 Small Seed Space in PRNG	0	1	0	0	0	1
CWE-352 Cross-Site Request Forgery (CSRF)	1	0	0	0	0	1
CWE-367 Time-of-check Time-of-use (TOCTOU) Race Condition	0	0	0	0	1	1
CWE-377 Insecure Temporary File	1	0	0	0	0	1
CWE-379 Creation of Temporary File in Directory with Incorrect Permissions	0	0	1	0	0	1
CWE-384 Session Fixation	0	0	1	0	0	1
CWE-385 Covert Timing Channel	0	1	0	0	0	1
CWE-400 Uncontrolled Resource Consumption	0	0	1	0	0	1
CWE-406 Insufficient Control of Network Message Volume	0	1	0	0	0	1
CWE-414 Missing Lock Check	0	0	0	0	1	1
CWE-425 Direct Request ('Forced Browsing')	0	0	0	0	1	1
CWE-454 External Initialization of Trusted Vars or Data Stores	0	0	0	0	1	1
CWE-462 Duplicate Key in Associative List	0	1	0	0	0	1
CWE-477 Use of Obsolete Function	0	0	0	0	1	1
CWE-488 Exposure of Data Element to Wrong Session	0	0	0	0	1	1
CWE-595 Comparison of Object References Instead of Object Contents	0	0	0	0	1	1
CWE-605 Multiple Binds to the Same Port	0	0	0	0	1	1
CWE-641 Improper Restriction of Names for Files and Other Resources	0	0	1	0	0	1
CWE-732 Incorrect Permission Assignment for Critical Resource	0	0	0	0	1	1
CWE-759 Use of a One-Way Hash without a Salt	0	1	0	0	0	1
CWE-760 Use of a One-Way Hash with a Predictable Salt	0	0	1	0	0	1
CWE-776 XML Entity Expansion	1	0	0	0	0	1
CWE-827 Improper Control of Document Type Definition	0	0	1	0	0	1
CWE-835 Infinite Loop	0	0	0	0	1	1
CWE-841 Improper Enforcement of Behavioral Workflow	0	1	0	0	0	1
CWE-941 Incorrectly Specified Destination in a Comm. Channel	0	1	0	0	0	1
CWE-943 Improper Neutralization of Special Elements in Data Query Logic	0	0	1	0	0	1

Table 1: Overview of our SecurityEval Dataset

for *demonstrative purposes* on how to use the dataset; the demonstration presented herein does not intend to be exhaustive.

3.1 Example: Using SecurityEval to Evaluate GitHub Copilot and InCoder

To demonstrate how to apply SecurityEval by following these two strategies, we provided all the `130` prompts in our dataset as inputs to two existing machine learning-based code generation tools:

- **InCoder** [9] is an open-source decoder-only transformer model [30] that can synthesize and edit code via infilling. We used the demo of the 6.7B parameter model available on Huggingface[2], where the number of tokens to generate is `128`, the temperature is `0.6` (default value)[3]. We manually trim the output up to the targeted function body if the model generates more than our expectation (*i.e.*, generating code after completing the function body). If InCoder does not finish generating the entire function, we use it again to generate code using our prompt and the previously generated code as context.
- **GitHub Copilot** [13] is a closed-source model behind a paywall from GitHub. The OpenAI Codex [6], an artificial intelligence model produced by OpenAI[4], powers GitHub Copilot. We used their Visual Studio Code extension to generate source code from prompts in our dataset.

Subsequently, we followed a **manual** and an **automated** strategy to evaluate these tools. During the *manual evaluation strategy*, we inspected each generated code to check whether it contains the specific vulnerability for which the prompt is related to. During the *automated evaluation strategy*, we analyzed the generated code using CodeQL [11] and Bandit [7], two static analyzers that can detect vulnerabilities and/or security smells. Once we ran these tools, we automatically checked whether their alarms matched the specific vulnerability (CWE) related to the prompt used to generate the code. For instance, if we used a prompt related to CWE-78 (OS

[2] https://huggingface.co/spaces/facebook/incoder-demo

[3] Temperature is a hyperparameter related to the probability of the model's output. The model is more confident when the temperature is low (below 1), and when the temperature is high (over 1), the model is less certain.

[4] https://openai.com

Command Injection), we checked the presence of CWE-78 in the generated code. Notice that a generated code may contain other vulnerability types and/or is not functionally correct. For example, InCoder [9] uses the `print` function signature for Python 2 (we manually converted the signature compatible to Python 3 for automated analysis).

Model	CodeQL	Bandit	Manual
InCoder [9]	20 (15.38%)	12 (9.23%)	88 (67.69%)
GitHub Copilot [13]	24 (18.46%)	14 (10.77%)	96 (73.84%)

Table 2: Evaluating InCoder [9] and GitHub Copilot [13] using SecurityEval

Table 2 presents the number of generated code snippets deemed vulnerable by relying on a manual or automated strategy. The numbers in the *CodeQL* and *Bandit* columns count the number of times in which a sample (associated with a specific CWE) was marked the generated code for that particular CWE (*automated strategy*). The *Manual* column contains the number of vulnerable generated codes after manually going through all the generated output and checking if the generated code contains the specific vulnerability (*i.e.*, the designated CWE for the sample.)

From these results, we observe that most generated code snippets contain insecure code (about 68% and 74% of code generated by InCoder and Copilot, respectively), which highlights the importance of evaluating generated code with respect to security concerns and not only functionality. Moreover, although an automated strategy decreases the time and effort in evaluating tools, they may not find all insecure code instances. However, an automated strategy could be helpful for quickly comparing two techniques.

4 THREATS TO VALIDITY

We consider four external sources for mining vulnerability examples to create the dataset in our work. We took the examples from the sources and modified them according to our task. CWE list [17] and CodeQL [11] are community-based and open-source project to enumerate common security vulnerability and detects them. Sonar Rules [23] from SonarSource provides documentation about the definition and rules for their static analyzers. Pearce *et al.* [20] is a peer-reviewed work. Though these external sources may introduce threats to our work, they are community-focused and widely used tools and sources for examples and definitions of common security weaknesses.

We used GitHub Copilot [13], a black box for generating source code. We also used the demonstration hosted on Huggingface for InCoder [9] instead of directly using the code for inference. However, the software and the model are open sources; hence, they can be verifiable/reproducible. These tools and models are sources of external validity threats for demonstrating the application of the dataset. Nevertheless, the application of this dataset is only for demonstration purposes.

This work focuses on Python, and one of our project goals is to establish the necessity to verify the machine learning model output from the security perspective. This establishment can be extended to other programming languages.

We manually crafted the examples from external sources and created additional examples to enrich our dataset. In addition, we manually checked the output from the model and tool after using our dataset by generating source code. These processes introduce internal threats to validity.

5 RELATED WORK

Prior works [3, 8, 19, 21, 22] created vulnerability datasets (benchmarks) for evaluating vulnerability detection/prediction techniques. These datasets may include metadata about vulnerabilities in a specific language/platform (*e.g.*, C/++ [8], Java [21], Android [3], *etc.*), their vulnerability types (CWE), and associated patches. Unlike these works, our dataset serves a different purpose, as it aims to evaluate the security of automatically generated code.

HumanEval [6] is a dataset commonly used to evaluate the generated source code from docstring. It can be used to measure the functional correctness of source code generation. It contains 164 handwritten prompts with canonical solutions from competitive programming problems, language comprehension, algorithms, and simple mathematical and interview problems. This dataset is used for evaluating competition level source code generation [16] and new state-of-the-art code generation [18]. However, it does not focus on the security aspect of the generated code. Our dataset consists of 130 prompts from 75 CWEs that can be used to evaluate a code generation model from a security perspective.

Pearce *et al.* [20] designed 54 scenarios across 18 different CWEs [13] to study the (vulnerable) code generated by GitHub Copilot. These scenarios focus on GitHub Copilot, whereas our dataset is a generalized one to use for any context-based source code generation model and tool. Our dataset is also rich with examples from 75 CWEs with 130 scenarios.

6 CONCLUSION & FUTURE WORK

Although a code generation model can help software engineers to develop software quickly, the generated code can contain security flaws. In this paper, we presented a dataset that is a diverse evaluation set for testing code generation models with respect to the presence of vulnerabilities. It has 130 Python code samples spanning 75 types of vulnerabilities (CWEs). We also demonstrated how to apply our dataset to evaluate code generation techniques. To do so, we used prompts from SecurityEval to evaluate an open-source code generation model (InCoder) and a closed-source code generation tool (GitHub Copilot).

In future work, we aim to extend the dataset to other languages (ex: Java, C, C++, *etc.*). Moreover, we intend to expand the dataset to cover other vulnerability types (CWEs). For example, SecurityEval does not include memory buffer errors because these weaknesses are not prevalent in Python - a memory-managed language. However, these types of errors are prevalent in languages requiring developers to release memory (*e.g.*, C/C++) manually.

REFERENCES

[1] 2022. Stack Overflow Developer Survey 2021. https://insights.stackoverflow.com/survey/2021 [Online; accessed 28. Aug. 2022].

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint

Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, UK) *(PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[4] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275* (2017).

[5] Stephen Cass. 2022. Top Programming Languages 2022. *IEEE Spectrum* (Aug. 2022). https://spectrum.ieee.org/top-programming-languages-2022

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[7] Bandit Developers. 2022. Bandit. https://bandit.readthedocs.io/en/latest/

[8] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[9] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. https://doi.org/10.48550/arXiv.2204.05999

[10] Yuexiu Gao and Chen Lyu. 2022. M2TS: Multi-Scale Multi-Modal Approach Based on Transformer for Source Code Summarization. *arXiv preprint arXiv:2203.09707* (2022).

[11] GitHub. 2022. CodeQL. https://github.com/github/codeql

[12] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2022. The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study. *IEEE Transactions on Software Engineering* (2022).

[13] GitHub Inc. 2022. GitHub Copilot : Your AI pair programmer. https://copilot.github.com

[14] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. CodeFill: Multi-token Code Completion by Jointly Learning from Structure and Naming Sequences. In *44th International Conference on Software Engineering (ICSE)*.

[15] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 150–162.

[16] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, et al. 2022. Competition-Level Code Generation with AlphaCode. https://doi.org/10.48550/ARXIV.2203.07814

[17] The MITRE Corporation (MITRE). 2022. Common Weakness Enumeration. https://cwe.mitre.org/

[18] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A Conversational Paradigm for Program Synthesis. https://doi.org/10.48550/arXiv.2203.13474

[19] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1565–1569.

[20] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 980–994. https://doi.org/10.1109/SP46214.2022.00057

[21] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 383–387.

[22] Sofia Reis and Rui Abreu. 2021. A ground-truth dataset of real security patches. *arXiv preprint arXiv:2110.09635* (2021).

[23] SonarSource S.A. 2022. SonarSource static code analysis. https://rules.sonarsource.com

[24] Joanna CS Santos, Anthony Peruma, Mehdi Mirakhorli, Matthias Galstery, Jairo Veloz Vidal, and Adriana Sejfia. 2017. Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of chromium, php and thunderbird. In *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 69–78.

[25] Joanna CS Santos, Katy Tarrit, Adriana Sejfia, Mehdi Mirakhorli, and Matthias Galster. 2019. An empirical study of tactical vulnerabilities. *Journal of Systems and Software* 149 (2019), 263–284.

[26] Mohammed Latif Siddiq, Shafayat Hossain Majumder, Maisha Rahman Mim, Sourov Jajodia, and Joanna CS Santos. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *22nd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)* (Limassol, Cyprus). IEEE.

[27] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991.

[28] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. 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*. 1433–1443.

[29] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 329–340.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. https://doi.org/10.48550/ARXIV.1706.03762