To be defined
Essential concepts
Names of programs, models etc
Important references

Transformer-based Vulnerability Detection in Code at EditTime: Zero-shot, Few-shot, or Fine-tuning?

Aaron Chan, Anant Kharkar, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy,
Alec Helyar, Eslam Kamal, Mohamed Elkamhawy, Neel Sundaresan

Microsoft
Redmond, USA

ABSTRACT

Software vulnerabilities bear enterprises significant costs. Despite extensive efforts in research and development of software vulnerability detection methods, uncaught vulnerabilities continue to put software owners and users at risk. Many current vulnerability detection methods require that code snippets can compile and build before attempting detection. This, unfortunately, introduces a long latency between the time a vulnerability is injected to the time it is removed, which can substantially increases the cost of fixing a vulnerability. We recognize that the current advances in machine learning can be used to detect vulnerable code patterns on syntactically incomplete code snippets as the developer is writing the code at *EditTime*. In this paper we present a practical system that leverages deep learning on a large-scale data set of vulnerable code patterns to learn complex manifestations of more than 250 vulnerability types and detect vulnerable code patterns at *EditTime*. We discuss zero-shot, few-shot, and fine-tuning approaches on state of the art pre-trained Large Language Models (LLMs). We show that in comparison with state of the art vulnerability detection models our approach improves the state of the art by 10%. We also evaluate our approach to detect vulnerability in auto-generated code by code LLMs. Evaluation on a benchmark of high-risk code scenarios shows a reduction of up to 90% vulnerability reduction.

KEYWORDS

Transformers, Software Vulnerabilities, Vulnerability Detection

1 INTRODUCTION

Despite development of many tools and best practices [3, 4, 12, 47], uncaught vulnerabilities persist in code[1, 2] and impact users and cost companies time and money[19]. In recent years, many approaches have been developed to detect vulnerabilities in software. These approaches range from traditional dynamic analysis [25, 28, 48], rule-based static analysis [9, 51], feature-based machine learning solutions [33], to more recent deep learning based solutions [8, 30, 41, 53].

Approaches that rely on dynamic analysis often suffer from low code coverage issues. The rule-based static analysis approaches typically involve manual efforts by experts to characterize and add new or evolved vulnerability patterns on a regular basis. Similarly, feature based machine learning approaches rely on hand-crafted features by human experts. More recently, deep learning-based approaches have emerged as an alternative which can learn vulnerability patterns automatically without expert involvement. However the majority of these Deep learning approaches require a complete source file [39], a complete function [23, 24, 41], or a complete code statement [11, 43] to analyze. As a result, to the best of our knowledge, the existing deep learning approaches cannot find vulnerabilities at *EditTime* while the developer is typing and the code is not yet syntactically correct or complete.

```
app.get('/:name', function(req, res, next){

// Query the account based on url parameters
connection.query('SELECT * FROM accounts WHERE
name="' + req.params.name + '"',
```

Figure 1: The best time to notify the developer about a *SQL-Injection* vulnerability is at *EditTime*, right after the developer has made the mistake.

The literature indicates that the cost of fixing a fault is positively correlated with the fault ignorance time (i.e the time when a vulnerability is injected to the time it is removed)[5, 6, 20]. Therefore, the developer should be notified about the fault in a timely manner, when the fault is relevant to the current programming task[22]. For example, in the code snippet shown in Figure 1, the best time to notify the developer about the SQL injection they have introduced is right at the end of the query, even though their code is not syntactically complete. The importance of interactively identifying bugs and vulnerabilities in code at *EditTime* has been recognized by prior work [50]. Our work extends this line of prior work by leveraging new advancements in deep learning to detect a variety of vulnerabilities in incomplete code snippets as the developer is typing.

To build our vulnerability detection solution, we collect more than 500K vulnerable code snippets from running static analyzers on open source repositories. We then develop six model variations using common learning approaches for pre-trained LLMs: zero-shot, few-shot and fine-tuning. We apply these learning techniques on CodeBERT[15] and two state of the art pre-trained LLMs provided by OpenAI: code-davinci-002, text-davinci-003. Our comparative evaluation of these variations shows that fine-tuning Code-BERT offers the best balance between precision and recall (precision of 59% and recall of 63%). However, both zero-shot and few-shot learning on recent text-davinci-003, which is an InstructGPT[35] based language model, offer better recall (78% and 75%) and slightly lower precision (47% and 49%). We discuss the benefits and challenges of each learning approach in terms of model size and performance.

We conduct two experiments using our best performing model. The first experiment compares this model with existing vulnerability detection approaches on four established benchmark datasets. We show that, in comparison with existing approaches, our approach improves recall up to 10% and precision up to 8%.

The second experiment studies the generalizability of our approach beyond manual code edits by developers, where we investigate the effectiveness of our approach in detecting vulnerabilities in automated code edits. With the recent wide-spread adoption of code completion tools [17, 45, 46], many future code edits are expected to be automatically generated by code LLMs. Prior work suggests that code LLMs make similar mistakes to developers [14], thus auto-generated code edits can have potentially severe security concerns. For example, in certain scenarios, up to 40% of the completions generated by code LLMs included vulnerable code patterns [38]. We evaluated our vulnerability detection model on a variant of the benchmark introduced by Pearce et al. [38]. Our evaluation shows that using our model yields 90% reduction in vulnerability rate.

Finally we discuss the deployment journey of our vulnerability detection model in a VSCode extension which has resulted in 80% reduction in rate of vulnerability in code edited with the extension.

This paper has the following contributions:

(1) we present a production-quality vulnerability detection system which can identify vulnerabilities in incomplete code snippets on the order of milliseconds and therefore serves developers at **EditTime**,

(2) we explore and discuss the benefits and challenges of three common learning approaches for pre-trained LLMs on the task of vulnerability detection: zero-shot, few-shot, and fine-tuning,

(3) we expand the benchmark introduced by Pearce et al. and make it available to the code vulnerability detection community as a future *EditTime* benchmark.

2 RELATED WORK

We discuss how our work builds upon and extends prior work in the area of Vulnerability Detection, Deep Learning methods for Vulnerability Detection, and Vulnerability Detection in auto-generated code.

2.1 Vulnerability Detection

There are two modalities of analyzing source code to search for vulnerabilities: dynamic analysis, in which a code snippet is executed and the stack trace examined for signatures of a specific vulnerability; and static analysis, in which the source code is analyzed without execution. One common method of dynamic analysis is fuzzing, in which a program is run on many different inputs to explore the space of program execution traces. Optimization techniques to efficiently search this program space are an active area of research [21, 31, 52]. However, fuzzing large code-bases by executing arbitrary code is infeasible, and has led to the development of static analysis techniques. Static analyzers search code-bases for semantic patterns of bugs or vulnerabilities[37]. CodeQL [18] is a popular scalable static analyzer for such pattern-based searches. CodeQL offers a query language to write rule-based queries for specific anti-patterns. We build upon this line of research by developing a vulnerability detection model that leverages state of the art deep learning approaches and therefore is capable of learning a variety of vulnerability types without involving human experts.

2.2 Deep Learning Vulnerability Detection

Since natural language and programming languages both possess sequential structure, model architectures with success in NLP, such as GRU, LSTM, and Transformer, have also shown promise in vulnerability detection [49]. One unique characteristic of source code, not shared by natural language, is the inherent graph structure of programs. VulDeePecker [30] extracts code gadgets from data dependency graphs and trains a BiLSTM classifier to detect vulnerabilities in a dataset of two different vulnerability types. SySeVR [29] builds on VulDeePecker by using both syntactic and semantic information to train vulnerability classifiers on a dataset of 126 vulnerability types. Other works leverage graph neural networks to supplement language models: IVdetect [27] uses GRU embeddings with a Feature-Attention Graph Convolutional Network (FA-GCN) to detect vulnerabilities in program dependence graphs. LineVul [16] extends IVdetect by leveraging pretrained CodeBERT [15] to perform line-level vulnerability localization in addition to method-level detection. Devign [53] learns embeddings of code representation graphs and trains a gated graph recurrent network to detect vulnerabilities for graph-level predictions. It evaluates this model on vulnerabilities from FFmpeg and QEMU. ReVeal [8] introduces a dataset of vulnerabilities from real-world software projects as a benchmark for vulnerability detection models. It demonstrates this benchmark using a gated graph neural network (GGNN) trained on code property graphs. Our research extends prior work on vulnerability detection by developing a neural vulnerability detection model for seven languages: JavaScript, Python, Go, Java, C++, C# and Ruby that is capable of detecting vulnerabilities at *EditTime*.

2.3 Vulnerability Detection in Auto-generated Code

Transformer-based models trained on source code have achieved state of the art results for code generation[10, 26, 34]. These models have unprecedented utility as developer-assistance tools, and are being deployed into products [17, 45, 46]. Since these code LLMs have been trained on large datasets of human-written code, their code outputs are increasingly likely to follow the patterns (or anti-patterns) of human developers. For example, [14] evaluated Codex on Java LeetCode questions and found that functionally incorrect completions from Codex share similar anti-patterns to incorrect code written by humans. Similarly, [38] examined Codex with regard to generating vulnerabilities and found that, in certain scenarios, Codex did indeed generate vulnerable code patterns. In another study, [44] showed that during functionality tests with the HumanEval dataset, GitHub Copilot generates certain CWEs in around 2% of cases. These studies suggest that code LLMs are likely learning to generate both bad and good code. Luckily, since vulnerable code patterns are rare in human code and therefore make up a small portion of training data, they are also rare in completions from Code LLMs. For example, a recent work studying the effect of code LLM assistance did not find any conclusive evidence that code LLMs are more likely to generate vulnerabilities than humans [42]. Nevertheless, given the high cost associated with code vulnerabilities, reducing vulnerable code patterns generated by code LLMs is essential in ensuring that their prolonged usage is safe, especially for novice programmers. To this end, our research yields a model

that can operate on syntactically incomplete code, and therefore detect vulnerabilities in auto-generated code snippets as well as the developer written code snippets at *EditTime*.

3 DETECTING VULNERABILITIES AT *EDITTIME*

To detect vulnerabilities at *EditTime* we develop six model variations using common learning approaches for pre-trained LLMs: zero-shot, few-shot and fine-tuning. Below, we first explain the training data we collected for our fine-tuning process. We then explain the model variants and their respective performances.

3.1 Data Collection

Our dataset consists of vulnerable code patterns detected in public GitHub repositories by the GitHub LGTM service. GitHub LGTM service runs CodeQL [18], a scalable static analyzer, on public GitHub repositories to identify a variety of vulnerable code patterns, including hard-coded credentials, SQL injections, and path injections. In this work, we select a subset of detected CodeQL issues that correspond to a set of "Common Weakness Enumeration" (CWE) [32] in each of seven languages: JavaScript, Python, Go, Java, C++, C#, and Ruby.[1] Table 1 shows the summary statistics of the curated dataset.

Table 1: Summary statistics of vulnerability detection training dataset

Model	Coverage	Vulnerable	Non Vulnerable
Javascript	70 CWEs	266,342	2,293,712
Python	37 CWEs	149,158	1,493,972
Go	29 CWEs	50,233	535,180
Java	44 CWEs	33,485	431,726
C++	32 CWEs	7,222	215,722
C#	54 CWEs	3,341	27,731
Ruby	19 CWEs	137	1,957

Each detected vulnerability contains the following:

- *Vulnerability title:* the type of detected vulnerability, corresponding to a CWE category
- *Message:* a detailed error message explaining the detected vulnerability
- *Location:* file path, line, and column where the issue starts and ends

3.2 Data Pre-Processing

Given the data we collected, the goal of our data pre-processing step was to synthesize the following triplets: (c_i, v_i, l_i) where c is a snippet of code that the model takes in as context, v is the block of code that is vulnerable, and l is the label for the vulnerability type. Our process for synthesizing training data is as follows: we first collect files with identified vulnerabilities from all 8,815 repositories in which these issues were detected. For each file, we extract the Abstract Syntax Tree (AST), and search for nodes of the tree that

---
[1]The source code for all CodeQL queries we ran can be found at https://github.com/github/codeql under the ql/Security folders.

contain complete scopes. Specifically, we look for statements (such as `if` and `export` statements), methods, declarations and clauses. For these blocks of code, if the code block contains a vulnerable code pattern, we randomly split the block at some character before the start of the detected vulnerability. Otherwise, if no vulnerability is detected, we randomly split the block at any point. This splitting process mimics a possible code state at *EditTime*. Therefore, models trained on this data should be able to detect a vulnerable block in a syntactically incorrect and incomplete code snippet. The first segment of the block is labeled as the "context" and the second segment as the "vulnerable block". Figure 2 shows an example of a context and vulnerable block pair. If the completion contains a vulnerability, the corresponding vulnerability type is assigned to the example as a label. This separation between context and vulnerable block forces the model to focus on the vulnerable block given the context. We used a 85/5/10 train-validation-test split at the repository level, such that the training set is comprised of 85% of the repositories, the validation set is comprised of 5% of the repositories and the test set is comprised of 10% of the repositories. We then removed any example in the training set that matches a test set example.

```
Context:
app.get('/:name', function(req, res, next){

    // Query the account based on url parameters
    con
Vulnerable Block:
    nection.query('SELECT * FROM accounts WHERE
    name="' + req.params.name + '"',
        function(err, rows, fields) {
            if (err) {
                next(err);
                return;
            }
            res.send(JSON.stringify(rows));
        });
})
```

Figure 2: A sample context and vulnerable block from our training data. In this example, the vulnerable block contains the *SQL-Injection* vulnerability.

3.3 Models

In order to detect vulnerabilities at *EditTime*, we develop six model variations using three learning approaches for LLMs: zero-shot learning, few-shot learning, and fine-tuning. We use these learning approaches on three pre-trained LLMs as our base:

- `code-davinci-002`: full-size Codex model which is trained on source code from GitHub.
- `text-davinci-003`: Codex model based on InstructGPT [36], using reinforcement learning from human feedback (RLHF)
- `CodeBERT`: transformer pre-trained trained on bimodal data (code and documents)[15]. Its architecture follows RoBERTa-base [13], with 12 layers, 12 attention heads, and a hidden size of 768.

Using three learning approaches on the above pre-trained models yields six models that are listed in Table 3. We explain the details of developing these models below.

3.3.1 Zero-shot Learning.
In zero-shot learning, we provide a pretrained model with a prompt that specifies our desired output. In our case, we prompt `codedavinci-002` and `text-davinci-003`. In order to obtain a suitable prompt for asking each model to detect vulnerabilities, we leverage the model's own definition of vulnerability detection. We first prompt the model to define the vulnerability detection task and use this task description in the final prompt for the task.

For `code-davinci-002` we prompt the model with *"# We run CodeQL security queries in order to "* on temperature 0.8 for four times and pick the top results. This yields the following prompt variations: *"identify potential security vulnerabilities"*, *"find potential security issues"*, *"find security vulnerabilities"*, and *"detect security vulnerabilities"*. We place these variations in a zero-shot prompt following the below template:

```
<phrase>
<comment> Code snippet
<code snippet>
<comment> Answer (Yes/No, explanation):
```

Here $<phrase>$ corresponds to one of the prompt phrasing variations above, $<comment>$ refers to a language specific comment indicator (e.g. "#" for Python) and $<code\ snippet>$ refers to the code snippet in question. Figure 3 shows a sample prompt following this template. In response to this prompt, the model outputs either Yes or No, followed by an explanation which can be used for observations and debugging purposes. We check for the presence of "Yes" or "No" to determine the model's decision. We refer to this zero-shot approach as **CodexZero**.

```
// identify potential security vulnerabilities
// Code snippet
const pg = require("pg");
const client = new pg.Client({
  host: "database.server.com",
  database: "mydb",
  port: 3211,
  user: "username",
  password: "password"
});
//Answer (Yes/No, explanation):
```
```

**Figure 3: A sample prompt created based on our template for zero-shot setting**

Similarly, for `text-davinci-003`, we first prompt the model four times at temperature 0.8 with the question, *"What would you accomplish by running CodeQL security queries?"*. We then ask the model to rephrase its response four times. This yields the following unique phrases: *"identify potential security vulnerabilities"*, *"spot any security weaknesses"*, *"detect any security risks"*, *"determine any security issues"*. We then try these variations in a zero-shot prompt following a similar template:

```
<phrase>
<code snippet>
Answer (Yes/No):
```

We check for the presence of "Yes" or "No" in the model's response to determine the model's decision. We refer to this zero-shot approach as **TextZero**.

3.3.2 Few-shot Learning.
Few-shot learning builds on zero-shot learning by providing example input-output pairs, in addition to the zero-shot prompt. For our study, we utilize the best performing prompt variations of `code-davinci-002` and `textdavinci-003` from our zero-shot learning in the same template format. We then prepend additional examples in the same template format before finally inserting the code snippet of interest and prompting the model for the answer.

To create the examples, we prompt each model with the phrase, *"Provide an example in <Language> of a code snippet that contains <Vulnerability Name> security vulnerability. Output the code only, do not include text:"* for each of the languages and types of vulnerabilities in Table 2. We prompt with this template three times for each vulnerability and language pair, yielding 150 vulnerable examples. We then prompt the model with *"Provide an example in <Language> of a code snippet. Output the code only, do not include text:"*, to retrieve non vulnerable samples of code. We manually evaluate each sample to ensure that they were vulnerable or non vulnerable as intended. In total, there were 297 examples. We refer to the models resulting from the above few-shot learning process with `code-davinci-002` and `text-davinci-003` as **CodexFew** and **TextFew**, respectively.

3.3.3 Fine-tuning.
For fine-tuning, we focus on pre-trained code LLMs only and fine-tune CodeBERT and `code-davinci-002`. For CodeBERT, we add a linear classification head to its BERT trunk in order to build a multi-class classification model. The inputs to the model are the context and vulnerable code block, separated by a special [SEP] token and bounded by a beginning of sequence [BOS] token and end of sequence [EOS] token

$$[BOS], c_1, c_2, ...c_n, [SEP], v_1, v_2, ...v_m, [EOS]$$

where $c_1, c_2, ...c_n$ denotes a sequence of n tokens of context code surrounding the vulnerability and $v_1, v_2, ...v_m$ denotes a sequence of m tokens of the vulnerable block of code. We distinguish between context and vulnerable block to enable the model to process any given complete or incomplete code snippet. Because our data is vastly imbalanced with less than 10% vulnerable examples, we employ an oversampling technique on the vulnerable examples while training: for each epoch, all vulnerable examples are retained while the non-vulnerable examples are rotated through so that each epoch contains 50% vulnerable and 50% non vulnerable examples. The model is trained with a standard binary cross entropy (BCE) loss. In the rest of this work, we refer to this model as **DeepDevVuln**.

Our second fine-tuned model is a fine-tuned version of `code-davinci-002` on 30,000 randomly sampled examples from our training set. We down-size our training data due to the cost of fine-tuning the model on the entirety of the training set. In this case, the context and vulnerable block are concatenated together, then a special classification token is appended to the end of the sequence. Because `code-davinci-002` is a GPT decoder model that does not inherently perform classification, we use next-token prediction as a proxy, i.e. the output is a special token that signifies vulnerable or not. The sequence is as follows:

$$[BOS], c_1, c_2, ...c_n, v_1, v_2, ..., v_n [CLS][VULN]$$

We refer to this variant as **CodexVuln**.

Table 2: Summary statistics of vulnerable issues gathered from Github PRs

Vulnerability	CWE	N
SQL Injection	89	45
Hardcoded Credentials	798	23
Code Injection	94	13
Path Injection	22	7
Clear Text Logging	312	5
Weak Cryptographic Algorithm	327	5
Incomplete URL Substring Sanitization	20	2

3.4 Model Variant Evaluation

To better understand the effect of model architecture and training choices, we compare our six model variants on a dataset we collect from GitHub pull requests.

3.4.1 Metrics. The majority of existing approaches treat vulnerability detection as a classification problem. Given a block of code, the model makes a binary decision on whether the code is vulnerable or not. Therefore, in our evaluations we also use the most common evaluation metrics for a classification task [40] which includes:

- **Precision**: which indicates the correctness of predictions and calculated as true positive / (true positive + false positive)
- **Recall**: which indicates the effectiveness of the prediction and calculated as true positive / (true positive + false negative).
- **F1-score**: which indicates balance between precision and recall and is defined as the geometric mean of the two.

Table 3: Performance of DeepDevVuln model on Github PR Vulnerabilities dataset.

Model	Precision	Recall	F1-Score
DeepDevVuln	58.87%	63.00%	**60.87**%
CodexVuln	**69.56**%	48.00%	56.80%
CodexZero	11.08%	**98.00**%	19.90%
TextZero	46.99%	78.00%	58.65%
CodexFew (8 examples)	23.91%	95.00%	37.70%
TextFew (6 examples)	49.01%	75.00%	59.29%

3.4.2 Dataset. To create this dataset, we collect all pull requests that contained *"fix <issue>"* in the pull request title for every combination of issue from Table 2 and language (i.e. JavaScript, Python, Go, Java, C++, C#, Ruby). To ensure that the retrieved pull requests were relevant to the target vulnerability, we only include the pull requests that contain both removal and addition of code and were less than 100 lines in length. This yields 283 pull requests. We then manually examine each pull request to ensure the vulnerabilities are legitimate.

Using this process, we gather a set of 100 vulnerable examples constituting the CWEs in Table 2. We added a set of 906 non vulnerable examples to this dataset. To collect the non vulnerable examples,

we randomly sample 150 files from repositories that CodeQL has scanned and has not detected an issue. We then randomly select chunks of non-overlapping code between 1 to 10 lines from these files to generate nonvulnerable examples. In total, the GitHub PR dataset contains 1,006 examples.

```
// Find if the code snippet has any security vulnerabilities
// Code snippet:
type groupSourceReq struct {
        Interface uint32
        Pad_cgo_0 [256]byte
// Answer (Yes/No, explanation):  Yes, the code snippet has security
// vulnerabilities.
// Explanation: The code snippet has a buffer overflow vulnerability.
// The variable "buf" is a fixed size array of 256 bytes.
```

Figure 4: A sample of CodexZero false positive due to over-reach

```
// Find if the code snippet has any security vulnerabilities
// Code snippet:
*(md++)        = (unsigned char)(t>>24);
*(md++)        = (unsigned char)(t>>16);
*(md++)        = (unsigned char)(t>>8);
*(md++)        = (unsigned char)(t);
// Answer (Yes/No, explanation):  Yes, buffer overflow.
// Explanation: The code snippet is vulnerable to buffer overflow.
// The code snippet is trying to write 4 bytes of data into the
// memory pointed by md.
// However, md is a pointer to a memory location that is not allocated.
```

Figure 5: A sample of CodexVuln false positive due to lack of context

3.4.3 Results. Table 3 shows the performance of our model variants on this dataset. For these six model variants, we can see that DeepDevVuln has the best performance with respect to F1 score. Every zero-shot and few-shot variant outperforms DeepDevVuln in terms of recall, however the precision of each variant is significantly lower. One reason might be that the zero-shot and few-shot models have a wider definition of what constitutes vulnerable code. For example, the code snippet presented in Figure 4 below does not have any explicit vulnerabilities, however, CodexZero considers the worst-case scenario that the code may be vulnerable to a buffer overflow in the future. Due to this tendency, it seems that CodexZero will raise an alert even if there is no explicit vulnerability present, which leads to a high recall score but low precision score.

Another example further explains CodexZero's high false positive rate. In Figure 5, we see that the model lacks context around the variable *md*, and explains that because it was never initialized, accessing the memory at that location could lead to a vulnerability.

This explanation extends, in part, to TextZero, which has a more balanced precision and recall. In TextZero's case, the prompt variation used had a significant effect: for example, the first prompt variation *"identify potential security vulnerabilities"* had similar results to CodexZero. However, using the phrase *"detect any security risks"* led to the result in Table 3. This may have encouraged the model to focus on the exact code snippet as-is, rather than to speculate about possible vulnerabilities.

Comparing few-shot and zero-shot results, we see the few-shot evaluations surpassed their zero-shot counterparts. One explanation is that the inclusion of examples gives the model a sense of what to expect in terms of code snippet length and area of focus. For instance, when the non-vulnerable examples include uninitialized variables and incomplete context, the model starts to ignore the worst-case scenarios explained above.

Finding the best number of examples was a matter of gradually increasing the number of examples prepended to the prompt from 1 to 9. We ran several trials and reported the best results. Among these trials, we observed that, especially for CodexFew, there is an apparent correlation between larger number of examples and the precision and recall achieved as seen in Figure 6. It may be that the exact types of examples selected have a major role in steering the model's output. A similar trend for precision, but not recall, was observed for TextFew, Figure 7.

4 EXPERIMENTS

Below, we explain our experiments designed to answer the following two research questions:

- **RQ1:** How effective is our vulnerability detection model compared to results from state of the art models on established benchmark datasets?
- **RQ2:** To what extent our proposed vulnerability detection model is effective in reducing the vulnerability rate of code LLMs?

We perform two experiments to answer these questions. The first experiment focuses on comparing our model against the state of the art vulnerability detection models on common datasets. The second experiment gauges the effectiveness of our model in reducing the vulnerability rate of code LLMs.

4.1 Experiment 1: Performance against Existing Approaches on Benchmark Datasets

In this section, we present the experimental evaluation of our vulnerability detection model against existing state of the art approaches on four widely used datasets. Table 4 summarizes the datasets used in this experiment.

4.1.1 Experimental Setup. The granularity of the data, quality of the data, and types of issues covered in each of the datasets in Table 4 is different from our training and test set. Therefore, we followed the approach of Chakraborty et al. [8], where we first remove duplicate examples from each of the datasets. We then fine-tune DeepDevVuln as a binary classification model on each dataset for 10 epochs. For each dataset, we use a standard 80%/10%/10% train/validation/test split, consistent with the baseline models.

4.1.2 Results. As shown in Table 5, our DeepDevVuln model has the best overall F1-Score for the majority of datasets. This means that our model demonstrates a good balance between precision and recall which is important for vulnerability detection. Additionally, our vulnerability detection model has 10 times fewer parameters than GPT2-Large or Codex, yet still achieves comparable precision and higher recall. Overall this results suggests that our approach allowed our model to adapt to the specific types of issues present

Figure 6: Trials varying examples of CodexFew

Figure 7: Trials varying examples of TextFew

in these datasets and leverage its knowledge gained through pretraining on our vulnerability dataset to improve upon the state of the art results.

4.2 Experiment 2: Model's effectiveness in reducing the vulnerability rate of code LLMs

In our second experiment, we evaluated the effectiveness of our vulnerability detection model in detecting the vulnerable code completions of four different code LLMs:

- `CodeGen-2B`: a transformer decoder model trained on natural language and code (C, C++, Go, Java, JavaScript, Python)
- `code-cushman-001`: smaller-size Codex model, trained on source code from GitHub
- `code-davinci-002`: full-size Codex model, trained on source code from GitHub
- `text-davinci-003`: Codex model based on InstructGPT [36], using reinforcement learning from human feedback (RLHF)

Table 4: Summary of common vulnerability detection datasets.

Dataset	# Programs	% Vuln	# Duplicates	Granularity	Description
VulDeePecker [30]	61,638	28.76%	33,786	Slice	It was obtained by preprocessing examples from the National Vulnerability Database (NVD) and the Software Assurance Reference Dataset (SARD) and consists of CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer) and CWE-399 (Resource Management Errors).
SeVC [29]	420,627	13.41%	188,030	Slice	An improvement over the VulDeePecker dataset, covering 126 different types of vulnerabilities and divided into four categories: API Function Call, Arithmetic Expression, Array Usage, and Pointer Usage.
ReVeal [8]	22,734	9.85%	351	Function	It tracks past vulnerabilities from the Linux Debian Kernel and Chromium projects. The dataset reflects real bug reports and has a realistic data imbalance, with only 10% of the examples being vulnerable.
FFMPeg+Qemu [53]	27,318	45.61%	60	Function	It consists of past vulnerabilities and their fixes from two open-source projects.

Table 5: Performance of DeepDevVuln model on Vuldeepecker, SeVC, Reveal, and FFMPeg+Qemu datasets.

Dataset	Model	Precision	Recall	F1-Score
VulDee-Pecker CWE 119	VulDeePecker	82.00%	91.70%	86.6%
	Thapa et al. CodeBERT	95.27%	95.15%	95.21%
	Thapa et al. GPT-2 Base	93.35%	93.56%	93.45%
	Thapa et al. GPT2-Large	95.74%	95.28%	95.51%
	Codex	**97.45%**	93.31%	95.33%
	DeepDevVuln	96.74%	**95.62%**	**96.18%**
VulDee-Pecker CWE 399	VulDeePecker	95.30%	94.60%	86.6%
	Thapa et al. CodeBERT	94.25%	95.29%	94.76%
	Thapa et al. GPT-2 Base	92.97%	94.99%	93.96%
	Thapa et al. GPT2-Large	**96.79%**	96.90%	96.84%
	Codex	96.69%	97.04%	**96.87%**
	DeepDevVuln	95.65%	**97.41%**	96.53%
SeVC	Thapa et al. (BERTBase)	88.73%	87.95%	88.34%
	Thapa et al. (GPT-2 Base)	86.88%	87.47%	88.34%
	Codex	82.26%	84.34%	83.29%
	DeepDevVuln	**95.56%**	**97.14%**	**96.35%**
ReVeal	Chakraborty et al.	30.91%	60.91%	41.25%
	Codex	45.04%	29.80%	35.87%
	CodeBERT	**48.95%**	35.35%	41.06%
	DeepDevVuln	41.00%	**61.00%**	**49.29%**
FFmpeg + Qemu	Chakraborty et al.	56.85%	74.61%	64.42%
	Codex	**63.22%**	55.64%	59.19%
	CodeBERT	62.94%	58.70%	60.74%
	DeepDevVuln	57.34%	**78.06%**	**66.11%**

We evaluated the extent to which our model detects the vulnerable code patterns produced by each LLM utilizing the benchmark created by Pearce et al[38]. This benchmark consists of scenarios to evaluate the likelihood of a code LLM generating vulnerabilities. These scenarios are constructed to specifically elicit a completion containing a particular vulnerability. Each scenario is associated with a particular CWE and includes a prompt code snippet and a corresponding CodeQL query. The prompt is used as input to a code LLM. The model-generated completion is appended to the prompt and this completed snippet is then analyzed using the provided CodeQL query. Completions that cannot be analyzed by CodeQL (due to syntactical errors) are considered invalid and excluded from the analysis. CodeQL marks the remaining valid completions as either vulnerable or clean.

We took the 29 Python scenarios developed by [38] and, following the same process, we added 11 new JavaScript scenarios covering 10 CWEs to the benchmark. Table 6 describes the scenarios we added using the same format as in [38]. "Rank" reflects the CWE ranking in the MITRE list if applicable. CWE-Scn refers to the scenario's identifier and associated CWE. All of these scenarios are written in JavaScript, originate from the public GitHub CodeQL documentation, and were evaluated with CodeQL.

Table 6: Javascript Scenarios covering 10 CWEs in Javascript

Rank	CWE-Scn.	Description
3	CWE-89	SQL Injection
4	CWE-20	Incomplete Url Substring Sanitization
8	CWE-22	Tainted Path
15	CWE-798	Hardcoded Credentials
25	CWE-94	Code Injection
35	CWE-601	Client Side Url Redirection
35	CWE-601	Server Side Url Redirection
40	CWE-312	Clear Text Storage of Sensitive Data
-	CWE-209	Stack Trace Exposure
-	CWE-327	Broken Cryptographic Algorithm
-	CWE-916	Insufficient Password Hash

For each scenario, a model can generate a varying number of valid or vulnerable completions. In the context of code LLMs, a developer may often see only a single completion for a given prompt. Therefore, we evaluate vulnerability rates at the level of scenarios (prompts): we count the number of scenarios that yielded at least one vulnerable completion. For example, suppose there are 10 scenarios and each model generates 5 completions per scenario. For each of the 10 scenarios, we run the corresponding CodeQL query on its 5 completions. Suppose that 9 scenarios have at least one syntactically valid completion. We consider the 9 scenarios with

valid completions and examine how many of the 9 have at least one vulnerable completion.

For each model, we generate 25 completions per scenario. We then run our vulnerability detection model on each completion and filter out the completions that our model detects as vulnerable. We then rerun the CodeQL queries from the benchmark on the remaining completions.

4.2.1 Results. The results of this vulnerability experiment are shown in Table 7. The first two columns corresponds to each code LLM acting alone, while the second two columns includes filtration from our vulnerability detection model. The vulnerability reduction rate is the percentage reduction in the vulnerability rate as a result of filtration.

As shown in the table, filtering vulnerable outputs results in a significant decrease in vulnerability rate. In particular, the vulnerability reduction rate is highest for `text-davinci-003`, which follows InstructGPT's method of reinforcement learning from human feedback (RLHF). RLHF is known to substantially improve the quality and naturalness of generated text. Therefore, `text-davinci-003` likely generates code that more closely resembles code written by real developers. Since DeepDevVuln is trained on developer-written code, it may be better able to detect vulnerabilities in outputs from `text-davinci-003` than other code LLMs.

5 DEPLOYING IN PRODUCTION

We deployed our model for detecting vulnerable code patterns in a VSCode extension with ~100K daily users. After each key stroke that a user writes, their incomplete code snippets are sent to our model for verification. To evaluate the effectiveness of our model, we collected and examined the JavaScript code snippets that were sent to our model in a three month period, from November 2022 to January 2023, for a total of ~6.7M code snippets. We chose to focus on JavaScript because it is the most popular language in VSCode.

In order to have a baseline for comparison, we ran all the CodeQL security queries for JavaScript on the collected code snippets. Overall, CodeQL detected ~1,284 vulnerable code snippets. However, this number is a lower-bound for the amount of actual vulnerable code snippets. CodeQL queries do not successfully run and detect issues in all of the code snippets, because CodeQL is designed to run on a set of complete files in a repository. Therefore, CodeQL's vulnerability detection rate drops significantly when executed on syntactically-incorrect code or incomplete code that is presented in a single file as opposed to the repository context. This drop impacts some scenarios more than others, depending on sensitivity of the query to syntax issues and the amount of context required by the query to detect a vulnerability. Of CodeQL's vulnerability detections, over 58% were from two scenarios which have simple CodeQL queries requiring less context to run successfully. In comparison, DeepDevVuln detections were more uniform. In fact, over 57% of DeepDevVuln detections were from SQL Injection, Code Injection, Client Side URL Redirect, Server Side URL Redirect, and Insufficient Password Hash scenarios. This is significant because JavaScript is a dominant language in both client- and server-side web development, and these classes should be more prominent in this domain. Yet, CodeQL detects these scenarios at a rate of less than 1 in 1,000,000. CodeQL's poor coverage and inability to detect

vulnerabilities on this production data highlights the need for deep learning based detection systems in live services.

For our evaluation, because CodeQL-detected issues are a lower-bound on the number of issues, we use them to measure recall. Instead of precision, we measure the positive rate (number of detected issues over number of all scanned issues). Figure 8 shows how our DeepDevVuln model performs on recall vs positive rate: it can achieve up to 90% recall with around 1% positive rate. While we optimized for recall for our extension, other applications can find the right balance between recall and positive rate based on their user scenario and feedback.

Figure 8: DeepDevVuln model's performance on recall vs. positive rate

Overall, we observed that DeepDevVuln's vulnerability reduction rate (i.e. the reduction in the rate of vulnerabilities present in developer's code) for JavaScript was 89.64%.

6 LESSONS LEARNED AND ONGOING WORK

In the process of building and deploying our model, we learned a few lessons that have guided our ongoing and future work on vulnerability detection.

First, the different learning methods we explored in this work come with trade-offs in model size, inference cost, predictive performance, and maintenance cost. Zero-shot and few-shot learning require sufficiently large models in order to effectively make predictions. Furthermore, predictive performance tends to improve with model size. For a constant model size, the inference cost of zero-shot learning is slightly higher than for fine-tuning, since a prompt must be constructed for each example; the cost for few-shot learning is even higher, since the system must fetch examples for each example. Our results show that a fine-tuning approach yielded the best prediction performance, allowing us to make accurate predictions without incurring high inference cost. However, in order to maintain a fine-tuned model, we must continually monitor and retrain the model on additional vulnerability types. Zero-shot and

Table 7: Vulnerability rate on scenario level for different baselines.

Approach	Without DeepDevVuln		With DeepDevVuln		Vulnerability Reduction Rate
	Valid Scenarios	Vulnerable Scenarios	Valid Scenarios	Vulnerable Scenarios	
CodeGen (6B)	7	7 (100.00%)	7	2 (29.00%)	71.00%
code-cushman-001	25	25 (100.00%)	19	5 (26.0%)	74.00%
code-davinci-002	26	24 (92.00%)	20	7 (35.0%	61.96%
text-davinci-003	27	21 (78.00%)	24	2 (8.0%)	89.74%

few-shot learning, on the other hand, would only require maintenance with regard to prompts and examples, rather than any further training.

Second, there is a trade-off between the size of a model and its response time. This work focuses on detecting vulnerabilities at *EditTime*, while a developer is writing code in an IDE. A large vulnerability detection model requires more time to make predictions, which can result in delayed response and the developer missing the vulnerability. In order to maintain low prediction latency, our vulnerability detection is based on the relatively small CodeBERT-base model and has under 100M parameters. As more powerful hardware is built to run large models and improve the inference time, we expect to be able to run large models in production settings in our future iterations.

Third, in many classification problems, a model's prediction threshold is used to create an appropriate balance between precision and recall. The balance is important because an effective production-ready fault detector must minimize churn and false positives [7]. High churn, where the issues raised vary from one version of the system to another, can cause significant friction with users due to a lack of consistency. False positives can similarly erode developer trust in the usefulness of a system, resulting in developers ignoring the tool. In our case, we do not have the ground truth of all vulnerable code snippets for our live evaluations, and therefore we cannot measure precision In this case, the analogous metrics are positive rate (the fraction of examples that the model predicts as positive) and recall. Our second lesson was in balancing these metrics for a production-scale vulnerability detection system. We tuned our threshold to maintain a 1% positive rate based on initial user's interactions and feedback. However more long-term studies and monitoring of these metrics are needed to better adjust the balance.

Finally we learned that we should periodically retrain our model to expand coverage as new vulnerability types are caught. When the common vulnerabilities are caught early on in the development process, users may start to notice the uncommon vulnerabilities and this may hurt their trust on detection tools overtime. Therefore, to address this challenge we have implemented a retraining pipeline where we constantly find datasets with new vulnerabilities to feed the pipeline and expand the coverage.

7 CONCLUSION

Code vulnerabilities continue to cost software companies and users. Detecting vulnerabilities in code at *EditTime* when the code is written by a developer or generated by a code LLM is essential to ensure the vulnerabilities are fixed at lower cost. Yet, the majority of current vulnerability detection tools do not detect vulnerabilities at *EditTime*. Our work closes this gap by presenting a vulnerability detection model that detects vulnerabilities on incomplete code snippets and therefore can be used to detect vulnerable code patterns at *EditTime* when code LLMs or developers write them. Our evaluation results showed that our model improves the state of the art detection approaches by 10% in recall and 8% in precision. In addition, our model reduces the vulnerability rate of code LLMs by >89%.

An immediate direction for future work is to expand our vulnerability detection model's coverage with adding new types of vulnerabilities to our training set. Another direction is to measure the long-term effect of our vulnerability detection model on the overall experience of developers who are using our VSCode extension. For example, measures can include the vulnerability reduction rate, whether the file under development resulted in a unit test failure, or whether a vulnerability was caught in the file after the *EditTime* (e.g. build time).

REFERENCES

[1] 2022. *CWE List Version 4.9.* https://cwe.mitre.org/data/index.html
[2] 2023. *NATIONAL VULNERABILITY DATABASE.* https://nvd.nist.gov/
[3] 2023. Secure Development. https://www.sei.cmu.edu/our-work/secure-development/index.cfm
[4] Shanai Ardi, David Byers, Per Hakon Meland, Inger Anne Tondel, and Nahid Shahmehri. 2007. How can the developer benefit from security modeling?. In *The Second International Conference on Availability, Reliability and Security (ARES'07)*. IEEE, 1017–1025.
[5] Boehm Barry et al. 1981. Software engineering economics. *New York* 197 (1981).
[6] Walter Baziuk. 1995. BNR/NORTEL: path to improve product quality, reliability and customer satisfaction. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. IEEE, 256–262.
[7] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.
[8] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).
[9] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 678–689.
[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei,

Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/ARXIV.2107.03374

[11] Min-je Choi, Sehun Jeong, Hakjoo Oh, and Jaegul Choo. 2017. End-to-end prediction of buffer overruns from raw source code via neural memory networks. *arXiv preprint arXiv:1703.02458* (2017).

[12] Microsoft Corp. 2023. *Microsoft Security Development Lifecycle.* https://www.microsoft.com/en-us/securityengineering/sdl/

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[14] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Improving automatically generated code from Codex via Automated Program Repair. https://doi.org/10.48550/ARXIV.2205.10583

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. https://doi.org/10.48550/ARXIV.2002.08155

[16] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction.

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

[18] Github Inc. 2021. CodeQL for research. https://securitylab.github.com/tools/codeql

[19] Dan Goodin. 2017. An NSA-derived ransomware worm is shutting down computers worldwide. https://arstechnica.com/information-technology/2017/05/an-nsa-derived-ransomware-worm-is-shutting-down-computer-world

[20] W Humphrey. 1995. A Discipline of Software Engineering Addison-Wesley. *Reading, Pa* (1995).

[21] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2022. CONFETTI: Amplifying Concolic Guidance for Fuzzers. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 438–450. https://doi.org/10.1145/3510003.3510628

[22] Lucas Layman, Laurie Williams, and Robert St Amant. 2007. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, 176–185.

[23] Tue Le, Tuan Nguyen, Trung Le, Dinh Phung, Paul Montague, Olivier De Vel, and Lizhen Qu. 2019. Maximal divergence sequential autoencoder for binary software vulnerability detection. In *International Conference on Learning Representations*.

[24] Young Jun Lee, Sang-Hoon Choi, Chulwoo Kim, Seung-Ho Lim, and Ki-Woong Park. 2017. Learning binary code with deep learning to detect software weakness. In *KSII the 9th international conference on internet (ICONI) 2017 symposium*.

[25] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 627–637.

[26] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814* (2022).

[27] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability Detection with Fine-Grained Interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 292–303. https://doi.org/10.1145/3468264.3468597

[28] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 533–544.

[29] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258.

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[31] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. 2021. Linear-time Temporal Logic guided Greybox Fuzzing. https://doi.org/10.48550/ARXIV.2109.02312

[32] T. M. C. (MITRE). 2022. CWE - Common Weakness Enumeration. https://cwe.mitre.org

[33] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*. 529–540.

[34] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[35] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155* (2022).

[36] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. https://doi.org/10.48550/ARXIV.2203.02155

[37] OWASP. 2021. "Source Code Analysis Tools.". https://owasp.org/www-community/SourceCodeAnalysisTools

[38] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. An Empirical Cybersecurity Evaluation of GitHub Copilot's Code Contributions. *CoRR* abs/2108.09293 (2021). arXiv:2108.09293 https://arxiv.org/abs/2108.09293

[39] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. 2015. Building program vector representations for deep learning. In *Knowledge Science, Engineering and Management: 8th International Conference, KSEM 2015, Chongqing, China, October 28-30, 2015, Proceedings 8*. Springer, 547–553.

[40] David MW Powers. 2020. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061* (2020).

[41] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 757–762.

[42] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2022. Security Implications of Large Language Model Code Assistants: A User Study. *arXiv preprint arXiv:2208.09727* (2022).

[43] Carson D Sestili, William S Snavely, and Nathan M VanHoudnos. 2018. Towards security defect prediction with AI. *arXiv preprint arXiv:1808.09897* (2018).

[44] Mohammed Latif Siddiq, Shafayat H Majumder, Maisha R Mim, Sourov Jajodia, and Joanna CS Santos. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. *Limassol, Cyprus, Oct* (2022).

[45] 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.

[46] TabNine. 2023. "AI assistant for software developers". https://www.tabnine.com/

[47] Blair Taylor and Shiva Azadegan. 2008. Moving beyond security tracks: integrating security in cs0 and cs1. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*. 320–324.

[48] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.

[49] Tongshuai Wu, Liwei Chen, Gewangzi Du, Chenguang Zhu, and Gang Shi. 2021. Self-Attention based Automated Vulnerability Detection with Effective Data Representation. In *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. 892–899. https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00126

[50] Jing Xie, Bill Chu, Heather Richter Lipford, and John T. Melton. 2011. ASIDE: IDE Support for Web Application Security. In *Proceedings of the 27th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '11)*. Association for Computing Machinery, New York, NY, USA, 267–276. https://doi.org/10.1145/2076732.2076770

[51] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. Spain: security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 462–472.

[52] Kunpeng Zhang, Xi Xiao, Xiaogang Zhu, Ruoxi Sun, Minhui Xue, and Sheng Wen. 2022. Path Transitions Tell More:Optimizing Fuzzing Schedules via Runtime Program States. https://doi.org/10.48550/ARXIV.2201.04441

[53] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).