6 Security Vulnerability Detection with Self-Instruct LLM Finetuning

Software security vulnerabilities allow attackers to perform malicious activities to disrupt software operations (i.e., security exploits). Since security exploits often occur during run-time, the independence of tests for LLM-based vulnerability detection shows promise.

you use run-time here but runtime later

Not sure if you present the motivation as "lead me to believe" or "leads me to believe"

The nuances of security vulnerabilities have been a pain-point for prior static analysis and machine learning vulnerability detection tools. The specific properties of security vulnerabilities lead me to believe that the attention-based LLMs can leverage these nuances for greater detection effectiveness. Importantly, the presence of rich, detailed vulnerability explanations available online leads me to believe a LLM can gain an understanding of how and why vulnerability can exist, and how that could lead to a security exploit.

As described in Chapter 3, Chapter 4, LLM can be a powerful tool for all stages of automated program repair: fault localization, patch generation, and plausible patch ranking. One opportunity from LLMAO is the capability of LLMs to perform security vulnerability detection without the usage of tests. However, a key limitation of LLMAO is the size of potentially vulnerable programs is limited by the maximum size of LLM context windows.

Security vulnerabilities often span across multiple functions or files. Unlike logic defects, security exploits take advantage of weaknesses in the data flow across an entire project. Furthermore, vulnerabilities correspond to specific exploits or attacks on a system, and the nuances of a vulnerability only unfold when accompanied by vulnerability *explanations*.

Note: be prepared to potential answer questions about this. Since new LLMs have context windows of 2M, is this still an issue?

To circumvent the LLM maximum context window issue while expanding our target programs for vulnerability detection, We trained LLMs using small snippets of code accompanied by important and relevant information of the code snippet, inferred by static analysis vulnerability explanations. As the final thrust of our dissertation work, we expanded our prior work to detecting vulnerabilities from larger programs, by detecting vulnerable functions across entire repositories instead of lines of code from single functions (i.e., LLMAO). We used static analysis of a potentially vulnerable program into trainable objectives for LLM-based vulnerability detection, and train LLMs on multiple dimensions of vulnerability information (i.e., multitask learning) in combination with dataflow-inspired graph neural networks (GNNs).

Multitask learning enables a model to learn shared knowledge and patterns simultaneously, typically leading to improved generalization and accuracy. My proposed approach is based on both recent advances in LLM research that enable fine-tuning on relatively small datasets, and the insights that (1) joint fine-tuning encompassing both code and vulnerability explanations can enhance performance compared to solitary code fine-tuning methods, and (2) most security vul-

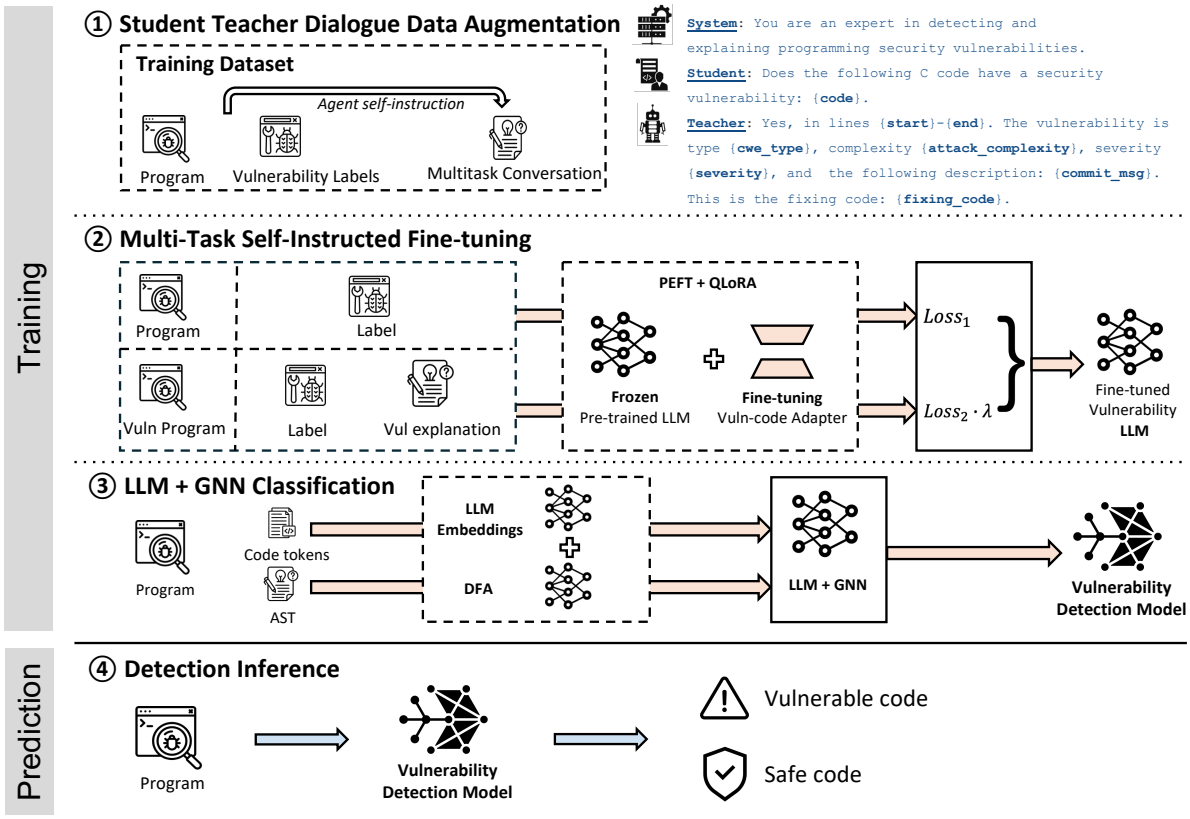

Figure 6.1: *MSIVD*'s architecture, which takes as training data a code snippet, its vulnerability label, and various human annotated vulnerability labels. *MSIVD* outputs a final vulnerability classification on unseen code snippets.

nerabilities entail specific and often subtle information flow, but training language models on either code or explanations alone will not capture key relations between values and data propagated through a potentially vulnerable program. Representing the program as a graph is therefore essential, in conjunction with the multi-task learning. We extract a subset of previously established security vulnerability datasets BigVul and PreciseBugs for model evaluation, and perform ablation studies on different combinations of pre-trained LLMs and security vulnerability explanations. Finally I propose our tool *MSIVD*: a multitask self-instruction LLM model for security vulnerability detection that trains on multiple types of vulnerability information.

6.1 MSIVD

Figure 6.1 provides an overview of MSIVD. The first three phases constitute training; the fourth phase, inference/vulnerability detection. The design rationale behind *MSIVD* is multi-faceted.

First, because security vulnerabilities are fundamentally complex, vulnerability reports and curated datasets include multi-dimensional information, e.g., textual categorization and explanations, implicated code, potential fixes, and static analysis reports. I aim to leverage these diverse types of *rich data* for training. During ① self-instruct dialogue-based data augmenta-

tion, *MSIVD* prepares a given dataset for fine-tuning by extracting vulnerability characteristics including type, description, and source code location. The second step, ② multi-task fine-tuning, then uses multi-task learning to fine-tune an LLM towards two learning objectives: (1) detecting a vulnerability and (2) providing an explanation that describes its characteristics. We focus on using *MSIVD* for vulnerability detection, and leave an evaluation (i.e., a human study) of the quality of produced explanations to future work. However, we predicate *MSIVD* on this type of multitask learning, including the explanation objective, because doing so can allow a model to simultaneously learn shared knowledge and patterns, improving generalization and accuracy. I overcome the small size of the available security vulnerability datasets by taking advantage of recent LLM advances that enable lightweight, parameter efficient [74], and quantized adapter level fine-tuning [138] suitable for smaller training datasets. Although these first two steps incorporate some code-level information, security vulnerabilities often entail specific, non-local, and subtle information flow. To reason about deeper semantic information, ③ LLM+GNN training, jointly trains the LLM with a GNN based on information flow data derived from the program's control flow graph. This step represents the program as a graph, allowing the fine-tuned LLM to reason directly about whether the values and data relations in a program indicate the occurrence of a vulnerability. In the ④ detection phase, given a program, the vulnerability detection LLM trained by *MSIVD* predicts whether a program contains a vulnerability.

6.1.1 Data Augmentation

To achieve security vulnerability detection incorporating vulnerability explanations, we first curated a custom dataset for both training and evaluating. The original dataset I use include samples of code snippets, and manually labelled classification on whether or not it includes a security vulnerability. The augmented dataset must allow the LLM to perform chain-of-thought, reasoning based on Self-instruct and Dialogue-policy-planned training. Therefore, the datapoints in our augmented vulnerability dataset must include a code snippet, its associated vulnerability label, CWE-type, a vulnerability description (e.g., how an attacker could exploit the vulnerability), and developer fix with fix location.

I transform a classification based security vulnerability dataset into a suitable dialogue, enabling a multi-round conversation format between a teacher and student. Inserting intermediate reasoning steps like this improves the ability of an LLM to perform complex reasoning. Each complete dialogue is a single training data entry. Embedded with the conversation is first a system prompt asserting that the teacher is "an expert in detecting and explaining programming security vulnerabilities", followed by a back-and-forth of questions and answers about the vulnerability and its associated information. Figure 6.2 shows a complete dialogue training data entry example. The teacher and student converse in three rounds of dialogue, each on a different aspect of the security vulnerability in a target code snippet. The first round of dialogue discusses the existence of the vulnerability; the second round, an explanation of why the code snippet has a vulnerability; and the third, which lines need to be changed to fix the vulnerability.

```
Round 0 = {
  role: "System",
  content: "You are an expert in detecting and locating security
      vulnerabilities, and can
    help answer vulnerability questions",
},
Round 1 = {
  role = "Student",
  content = f"Does the following code have any security vulnerabilities
      : {code_snippet}",

  role = "Teacher",
  content = f"Yes. The following code has a vulnerability type {
      cwe_type}.",
},
Round 2 = {
  role = "Student",
  content = f"What is the vulnerability description?",

  role = "Teacher",
  content =f"The vulnerability is:{commit_msg}",
}
Round 3 = {
  role = "Student",
  content = f"Locate the lines that are vulnerable and should be
      repaired.",

  role = "Teacher",
  content = f"The code is vulnerable at lines {vuln_lines}, with the
      following fix: {fixing_code}",
}
```

Figure 6.2: A single training data entry for the vulnerability detection multi-task fine-tuning. The dialogue rounds follow the four types of labelled data: vulnerability classification label, description, type, and repair lines.

6.1.2 Model setup

I train two models for *MSIVD*. The first performs a sequence-to-sequence fine-tuning of a selected pre-trained LLM, using our multitask self-instruct approach. The second performs a sequence-to-classification training loop that outputs a binary classification label (i.e., if the sample is vulnerable or not), which we build on top of DeepDFA's GNN architecture. The second model takes the final hidden states from the frozen in place first model. We refer to the tool using both models as *MSIVD* throughout evaluation; the tool consisting only of the first model, without the GNN architecture, as *MSIVD-*. *MSIVD-* converts the first model into a sequence-to-classification model directly, using a single `linear` layer. For the initial pre-trained model, we use CodeLlama-13B-Instruct [106], which is the 13 billion parameters instruction tuned version of CodeLlama. CodeLlama released 4 model size versions, from 7B to 70B. Due to limited computing and VRAM, we chose the 13B version. Table 6.1 shows the hyperparameters used for both

Table 6.1: Hyperparameters for multitask self-instruct fine-tuning, and LLM-GNN combined vulnerability detection model training.

Hyperparameter	Multitask FT	LLM+GNN
Initial Learning Rate	1e-5	1e-6
Model Dimension	4096	4352
Context Window	2048	2048
Layers	8	11
Batch Size	4	4
Epochs	10	5

models. The 4352 model dimension from the LLM-GNN model is a result of concatenating the fine-tuned LLM (4096) with the GNN model (256). Similarly, we add the output layers of LLM with the GNN to form $8 + 3 = 11$ layers. For batch size, we use 4 to fit CodeLlama 13B onto a single RTX 8000 GPU. However, other GPUs with more VRAM could employ higher batch sizes for greater efficiency. All experients used an Intel(R) Xeon(R) 6248R CPU @ 3.00GHz running Debian GNU/Linux 1 and two Nvidia Quadro RTX 8000 GPUs.

6.2 Evaluation and Results

In this section, we present the results evaluating *MSIVD*'s performance by answering three research questions:

RQ1: How effective is *MSIVD* at finding vulnerabilities on an established dataset?

RQ2: To what extent can *MSIVD* generalize to known-unseen vulnerabilities?

RQ3: How does each component of *MSIVD*, or vulnerability type, impact performance?

All results presented in this section were obtained using an Intel(R) Xeon(R) 6248R CPU @ 3.00GHz running Debian GNU/Linux 1 and two Nvidia Quadro RTX 8000 GPUs.

6.2.1 Setup

For classification, we convert the existence of a vulnerability into binary labels. To characterize classification effectiveness for an entire dataset, we use F1, precision, and recall, following the same metrics used for the prior work [34, 67, 72, 113].

Security vulnerability detection has a rich research history long pre-dating recent advances in machine learning. We focus our empirical comparison on work that uses machine learning for static vulnerability detection, as the recent results are especially promising, including compared to older, non-ML-based approaches. We compare *MSIVD* to techniques that target general vulnerability detection (that is, not restricted to particular rule-sets or vulnerability types) at the same granularity level (the function level),[1] and that either provide replication packages that can

[1]This granularity choice excludes several recent techniques, like Vuddy [50], VulChcker [89], and LineVD [38]; these are statement-level.

Table 6.2: Vulnerability prediction effectiveness on the Bigvul dataset. VulDeePecker, SySeVR, Draper, and IVDetect performance are from ref [67], and CodeBERT and CodeT5 performance from ref [113].

Type	Technique	F1	Precision	Recall
Random	Random	0.11	0.06	0.50
Non-LLM	VulDeePecker	0.12	0.49	0.19
	SySeVR	0.15	0.74	0.27
	Draper	0.16	0.48	0.24
	IVDetect	0.23	0.72	0.35
	DeepDFA	0.67	0.54	0.90
LLM	CodeBERT	0.21	0.68	0.13
	CodeT5	0.46	0.56	0.39
	CodeLlama	0.74	0.85	0.63
	LineVul	0.81	0.86	0.78
LLM + GNN	CodeT5 + DeepDFA	0.79	0.85	0.71
	LineVul + DeepDFA	0.88	0.88	0.89
	MSIVD	**0.92**	**0.93**	**0.91**

be applied to *BigVul*, or include evaluation results on *BigVul* in the original publication. Our chosen baselines cover several categories:

- ***Non-LLM*** *deep learning-based vulnerability detection tools* VulDeePecker [72], SySeVR [127], Draper [107], IVDetect [67], and DeepDFA [113].
- ***LLM****-based approaches*, including (1) the vulnerability detection capabilities of open-source, pre-trained LLM models CodeBERT [30], CodeT5 [127], and CodeLlama [106] (code pre-trained version of Llama-2), and (2) LineVul [34], which trains an additional sequence-classification model on top of a pre-trained LLM. Although LineVul originally uses CodeBERT and RoBERTA [77] as its pre-trained LLM, we customize it to use the same pre-trained model (CodeLlama-13B) as *MSIVD* and train for the same number of epochs (5). Otherwise, differences between *MSIVD* and LineVul could be a due to differences in pre-trained model effectiveness, rather than approach.
- ***LLM + GNN*** *combined techniques* using DeepDFA's replication package to combine their GNN embeddings with our fine-tuned model, as well as any HuggingFace pre-trained model directly. We train the same number of epochs (5) as *MSIVD*.
- ***Random*** baseline that predicts whether a sample is vulnerable with a probability of 0.5, to ground precision, recall, and F1, whose performance is sensitive to the underlying data distribution (and our datasets are imbalanced).

We are missing the description of the datasets. You also had more information in Section IV. Evaluation Setup from the paper that you could add here.

Our chosen baselines represent the state-of-the-art of vulnerability detection models [113], and all can be evaluated on the the *BigVul* dataset. Prior techniques relying on extant program analysis results cannot be trivially evaluated on the *PreciseBugs* dataset; we therefore only evaluate the best-performing LLM-based techniques on it.

6.2.2 Results

6.2.3 RQ1: Effectiveness on established dataset

Table 6.2 shows the effectiveness of *MSIVD*, and all baselines, on the *BigVul* dataset. DeepDFA's data flow analysis-based GNN technique outperforms prior non-LLM techniques, with a F1 score of 0.67, largely due to its high recall of 0.9 (correctly identifying 90% of the vulnerable code samples). However, Table 6.2 also shows that *all* LLM approaches other than CodeBERT outperform all non-LLM approaches. That is, LineVul, without any insights from program analysis, surpasses all state-of-the-art program-analysis-based deep learning tools, although it is even more effective when combined with DeepDFA's GNN layers. Model knowledge from static analysis provides limited improvement for more powerful LLMs like CodeLlama; this knowledge more clearly benefits more dated LLMs like CodeT5 (i.e., F1 score improving from 0.46 to 0.79). Overall, *MSVID* achieves F1 of 0.92, precision 0.93, and recall 0.91, outperforming all baselines on all metrics. These results demonstrate that the different aspects of vulnerability explanation can indeed improve a pre-trained LLM's vulnerability detection accuracy. However, *MSVID* only shows incremental improvements over LineVul + DeepDFA, as compared to LineVul + DeepDFA's larger improvements over all non-LLM tools. The largest improvements on vulnerability detection with the *BigVul* dataset comes from the underlying LLM itself.

RQ1 Summary

LLM-based techniques outperform non-LLM techniques on the *bigvul* dataset. *MSVID* outperforms prior all state-of-the-art approaches, with an F1 score of 0.92. The incremental improvements of adding either GNNs or fine-tuning suggests that the underlying pre-trained LLM is capable of effective vulnerability prediction based on code tokens alone.

6.2.4 RQ2: *MSIVD* effectiveness on unseen vulnerabilities

Table 6.2 shows *MSIVD*-'s performance on the *PreciseBugs* dataset, comparing to LineVul (the best-performing prior technique on *BigVul*). *MSIVD* shows a larger improvement over LineVul on the newer dataset (F1 of 0.48 to LineVul's F1 of 0.31) as compared to on *BigVul*. This demonstrates the relative effectiveness of our fine-tuning approach on unseen vulnerabilities.

We sought to understand why LineVul with CodeLlama as the underlying model performs so much worse on *PreciseBugs* than on *BigVul*. One clue lies in CodeLlama's effectiveness on our evaluation data on its own: Table 6.2 shows that CodeLlama achieves an F1 score of 0.74 on *BigVul*, but only 0.22 on *PreciseBugs*. Without any additional vulnerability classification training or sequence-to-sequence fine-tuning, CodeLlama already beats most prior non-LLM techniques.

More telling evidence emerges from the training loss function for CodeLlama using our multitask fine-tuning method. A deep-learning model's loss curve describes how closely a model's predictions are to the ground truth. We expect loss to generally decrease as training epochs increase; lower loss means better prediction. Figure 6.3 shows the loss curves of our training approach on the *BigVul* dataset with (i.e., multitask fine-tuning) and without explanations (i.e., label-only fine-tuning). The loss curve on fine-tuning *BigVul* with explanation approaches 0.2 in 400 steps (2 epochs, roughly 16 hours of training time). In contrast, the loss curve on

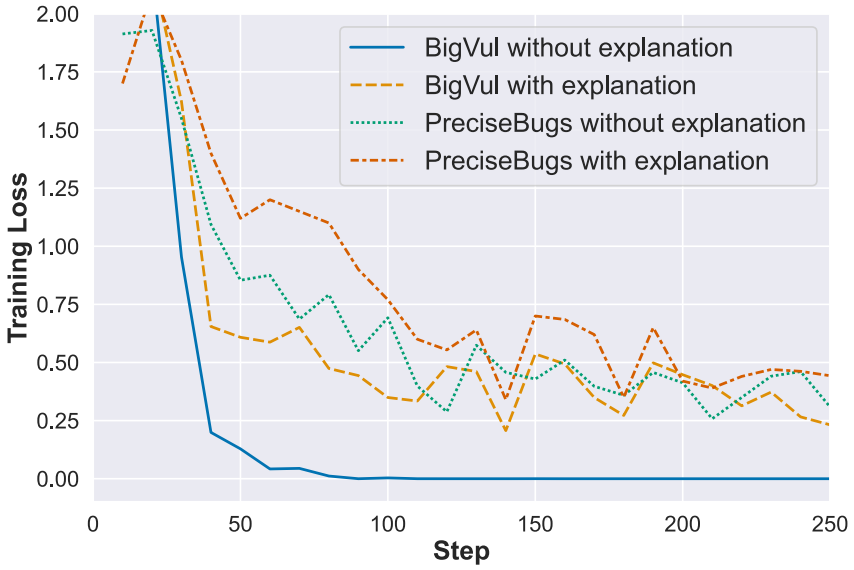

Figure 6.3: Loss curve of *MSIVD* fine-tuning CodeLlama for *BigVul* and *PreciseBugs*. Lower loss indicates model predictions closer to the ground-truth labels. Near-zero loss, with minimal training, indicates over-fitting [126]. Note that the pre-trained LLM (CodeLlama) shows obvious over-fitting on *BigVul* without explanations, motivating the necessity for new labelled datasets for model evaluation.

fine-tuning *BigVul* without explanations approaches 0.2 in 50 steps (1/4 of an epoch, roughly 2 hours of training time). Near-zero loss, with minimal training, is a strong indication of data leakage and overfitting. Here, the pre-trained LLM (CodeLlama) shows clear over-fitting (i.e., prior memorization of the dataset) on *BigVul* without explanations. Adding explanations includes more information, and adding non-binary labelling naturally induces noise, explaining why fine-tuning with explanations is not as overfit. Note that we observe the same loss curve behavior on the *Devign* dataset, using the same setup (not shown). This suggests that much of the accuracy of LLM-based techniques is likely arises from dataset memorization. This supports the importance of evaluating LLM-based approaches on labelled vulnerabilities collected after the selected LLM's training data cut-off date. Our results also demonstrate the value of fine-tuning LLMs using a multitask approach which, even on previously seen data, inserts a useful degree of randomness in learning.

RQ2 Summary

Neither CodeLlama alone nor prior LLM-based vulnerability detector baselines generalize well to the unseen *PreciseBugs* dataset. Training loss curves suggest that CodeLlama has likely memorized the *BigVul* dataset. While *MSIVD* is therefore more effective on *BigVul* than on *PreciseBugs* vulnerabilities, it better generalizes to the recently released *PreciseBugs* dataset than the prior baselines.

6.2.5 RQ3: Ablation study

Setup. To answer RQ3, we evaluate *MSIVD* under four settings and evaluate their performances on *BigVul* and *PreciseBugs*. First, we use the underlying pre-trained LLM directly for prediction (as in *RQ1*). We then use a fine-tuned version of *MSIVD*, but without any vulnerability explanations in its training data (label-only FT). Finally, we include vulnerability explanations in a single round of self-instruction fine-tuning (single-round SIFT) and multiple rounds of self-instruction fine-tuning (multi-round SIFT, which corresponds to *MSIVD-*). For *BigVul*, we also add the GNN adapter layers (multi-round SIFT + GNN, which corresponds to the full version of *MSIVD*).

We additionally evaluate our tool on specific vulnerability types within *PreciseBugs*, (training/evaluating on single vulnerability types). We choose the three most common types from our *PreciseBugs* dataset: buffer error (27.3% of *PreciseBugs*), resource error (21.2% of *PreciseBugs*), and input validation error (13.6%).

Results. Table 6.3 shows results on both the *BigVul* and *PreciseBugs* datasets. As discussed in Section 6.2.4, CodeLlama already performs well at detecting *BigVul* vulnerabilities. Training a separate model without using agent self-instruction slightly improves effectiveness, with a F1 score of 0.81 (+0.07 above the pre-trained F1 score of 0.74) for *BigVul*, and a F1 score of 0.33 (+0.1 above pre-trained) on *PreciseBugs*.

Surprisingly, we find that fine-tuning on only the vulnerability label and none of the explanations actually performs worse than using a pre-trained model directly for vulnerability classification (0.71 F1 for fine-tuned CodeLlama, and 0.74 F1 for pre-trained CodeLlama on the *BigVul* dataset). Our findings are consistent with those of Yusuf et al. [145], who observed that instruction-based fine-tuning may not always enhance performance, especially across a dataset of diverse CWEs. The shift from sequence-to-sequence fine-tuning to the sequence-classification training within a small dataset may simply include more noise, reducing classification performance.

Fine-tuning with both code and vulnerability explanations with the multitask agent setup (*MSIVD*) yields the highest vulnerability detection on both *BigVul* and *PreciseBugs*. We also see that training with multi-round SIFT yields higher F1 scores than single-round SIFT (a F1 improvement of 0.09 for *BigVul*, and 0.02 for *PreciseBugs*), which is consistent with prior work on LLM instruction-prompting [53]. Finally, we observe that the additional GNN (multi-round SIFT + GNN) provides an additional 0.02 F1 on top of multi-round SIFT for the *BigVul* dataset. The incremental improvement from the addition of GNN shows that CodeLlama already makes accurate predictions based on prior knowledge on the *BigVul* dataset, as previously discussed in Section 6.2.4.

Table 6.3 shows that training and evaluating on single vulnerability types improves F1 scores as compared to the entire dataset, but by trading off higher precision for lower recall. These results further corroborate that the LLM-unseen vulnerabilities in the newer *PreciseBugs* dataset are more difficult for any language model to detect. However, our results also indicate that training with a multi-round self-instruct format on a dataset with both label and explanation produces considerable improvements over pre-trained models alone, and substantiate the value of the individual components of *MSIVD*'s design.

Table 6.3: *MSIVD* ablation study. "Pre-trained" uses the underlying LLM directly for vulnerability detection. "Label-only fine-tuned" (FT) performs single-task fine-tuning on the vulnerability classification labels. "Single round self-instruct fine-tuned (SIFT)" trains the LLM without the agent explanation multi-round dialogue. "Multi-round SIFT" uses multi-task agent-dialogue fine-tuning (*MSIVD*$^{-}$). "Multi-round SIFT + GNN" adds the GNN adapter layer and corresponds to the full version of *MSIVD*.

Dataset	Technique	F1	Precision	Recall
BigVul	Pre-trained	0.74	0.85	0.55
	Label-only FT	0.71	0.77	0.66
	Single-round SIFT	0.81	0.86	0.61
	Multi-round SIFT	0.90	0.91	0.87
	Multi-round SIFT + GNN	0.92	0.93	0.91
PreciseBugs	Pre-trained	0.22	0.16	0.35
	Label-only FT	0.31	0.43	0.25
	Single-round SIFT	0.33	0.46	0.25
	Multi-round SIFT	0.48	0.4	0.57
PreciseBugs Vuln. Type	*MSIVD*minus Input	0.46	0.49	0.44
	*MSIVD*minus Resource	0.58	**0.63**	0.51
	*MSIVD*minus Buffer	**0.59**	0.62	**0.57**

RQ3 Summary

Further training a code LLM on vulnerability-specific code and labels improves detection effectiveness. Fine-tuning an LLM without vulnerability explanations actually reduces effectiveness as compared to the pre-trained model alone. Multitask fine-tuning with all included vulnerability explanations achieves the highest detection effectiveness, especially with multiple rounds of self-instruction. Finally, selecting specific vulnerability types for both training and evaluating yields higher F1 scores, but with a trade-off of lower recall due to the smaller data size.

6.3 Conclusion

Automatically detecting software security vulnerabilities is a rich and longstanding problem. Recent advances in ML have enabled techniques to combine program analysis with deep learning or LLMs. Meanwhile, the relatively small curated security vulnerability datasets provide rich additional information that prior work has left unexplored. We bridge this gap by introducing a self-instruct based multitask fine-tuning model to learn vulnerability classification based on both program code and vulnerability explanations. We further include information from data flow analysis, and build a light-weight GNN adapter based on a program's call graph to achieve simultaneous transfer-learning between LLM and GNN. Our tool surpasses prior state-of-the-art results on established vulnerability datasets. Furthermore, because of the risk (and evidence) of LLM data contamination, we collect a novel vulnerability dataset with evaluation and test sam-

ples exclusively filtered to be vulnerabilities identified past our pre-trained code LLM's training cutoff, and show that our technique outperforms prior work on that dataset as well.

7 Node.js Vulnerability Detection with Program Analysis and pretrained LLMs

Add citation to provenance graphs. You may want to explain what provenance graphs are in more detail when talking about NodeMedic-FINE.

As discussed in Chapter 6, we aim to improve the vulnerability detector from Chapter 3 for larger, real-world repositories. Chapter 6 extends *LLMAO* with *MSIVD*, which addresses the key limitation of context window sizing. *MSIVD* leverages the data flow analysis (DFA) of a program into trainable objectives for LLM-based vulnerability detection. DFA tracks how data moves through program variables by capturing control flow and data dependencies. However, not all information passed across a repository is represented by variables. A more specific static analysis technique is the creation of *provenance graphs*. Provenance graphs track data origins and transformations, and focuses on capturing relationships between data artifacts and processes. Provenance graphs investigate the "why" and "how" of data transformations, which are necessary for uncovering certain exploits. In this work, we further extend our previous vulnerability detector with provenance graphs, which are constructed through dynamic taint analysis that capture the complete history of operations applied to tainted values during a program's execution. Each node in a provenance graph represents a taint-related operation or value, while edges illustrate the flow of data between these nodes.

You to explain what are ACI and ACE attacks.

The popular Node.js ecosystem has become an attractive target for attackers. Two of the most serious vulnerabilities are Arbitrary Command Injection (ACI) and Arbitrary Code Execution (ACE), which allow attackers to execute malicious commands or code on the system that runs the application. Prior work such as NODEMEDIC-FINE used program synthesis to generate proof-of-concept exploits for Node.js packages for vulnerability confirmation. However, there are many instances where no exploit is produced, resulting in a high number of false positives that require manual analysis.

This work investigates whether ML-based approaches can predict ACE and ACI vulnerabilities in Node.js packages. We present a large-scale dataset of over 2,000 package-level vulnerabilities with human-reviewed labels. We conduct a comprehensive evaluation of approaches based on large language models (LLMs), program analysis extended with ML approaches, and hybrid ML methods. We integrate the vulnerability detection engine of a prior program analysis tool, NODEMEDIC, with Graph Neural Networks, Random Forest, XGBoost, Logistic Regression, SVM, and LLMs to reduce false negatives.

Our results indicate that both traditional machine learning models and LLMs when combined with program analysis tools can predict Node.js package exploitability with high accuracy. Our work provides insights into the effectiveness and challenges of using LLMs for vulnerability detection in JavaScript packages.

Table 7.1: Overview of the dataset splits used in the evaluation. The table displays the total number of packages in each split (*train*, *validate*, and *test*), along with the number of vulnerable packages in each split, categorized into ACE and ACI. Numbers in parentheses indicate the count of vulnerable packages within each split.

Split	Total (Vuln)	ACE (Vuln)	ACI (Vuln)
train	1640 (1173)	335 (290)	1,305 (883)
validate	205 (140)	45 (38)	160 (102)
test	206 (152)	40 (32)	166 (120)
total	2,051 (1,465)	420 (360)	1,631 (1,105)

7.1 Approach

There are many approaches in Figure 7.1. You explain them in different subsections . I would put a reference in the text to where they are explained (Section 7.1.2., etc).

In this work, we conduct a comprehensive evaluation of LLM-based methods, program analysis-based methods, ML-enhanced program analysis methods, and a hybrid approach that combines LLMs, NODEMEDIC-FINE, and GNNs for detecting vulnerabilities in JavaScript code. A summary of our approaches is shown in Figure 7.1. More concretely, we implement the following approaches with machine learning predictors that can help reduce the effort required for manual confirmation of package exploitability. (1) LLMs that do not rely on program analysis tools. (2) NODEMEDIC-ML that applies various ML models (Random Forest, XGBoost, Logistic Regression, and SVM) to predict vulnerabilities based on the provenance graph output from the dynamic taint analysis tool NODEMEDIC-FINE. (3)NODEMEDIC-GNN, which is setup similarly as NODEMEDIC-ML but uses Graph Neural Networks. (4) NODEMEDIC-GNN-LLM, which combines the embedding layers of the GNN model and the pre-trained language models (LMs).

7.1.1 Dataset

The dataset used for our evaluation consists of 2,051 npm packages in which NODEMEDIC-FINE identified potential vulnerable calls to sinks. These packages were selected from a total of 33,011 npm packages analyzed for ACI and ACE vulnerabilities by NODEMEDIC-FINE and shared by the authors of NODEMEDIC-FINE.

Of the 2,051 potentially vulnerable npm packages, 1,465 have been confirmed to be vulnerable with working exploits. Of these, 728 were automatically confirmed by NODEMEDIC-FINE, while 737 were confirmed manually.

Since LLM methods, the LLM-GNN hybrid method, and ML-enhanced NODEMEDIC-FINE methods require parts of the data for training and validation, we randomly divided the 2,051 package dataset into three subsets (train, validation, and test) in an 8:1:1 ratio. The training set includes 1,640 packages, the validation set includes 205 packages, and the testing set includes 206 packages. For models requiring training, the training set is used to train the model, while the validation set is used to select the best model model before evaluation. All models are then evaluated on the same testing set to ensure consistent performance reporting. Table 7.1 provides a detailed overview of these dataset splits.

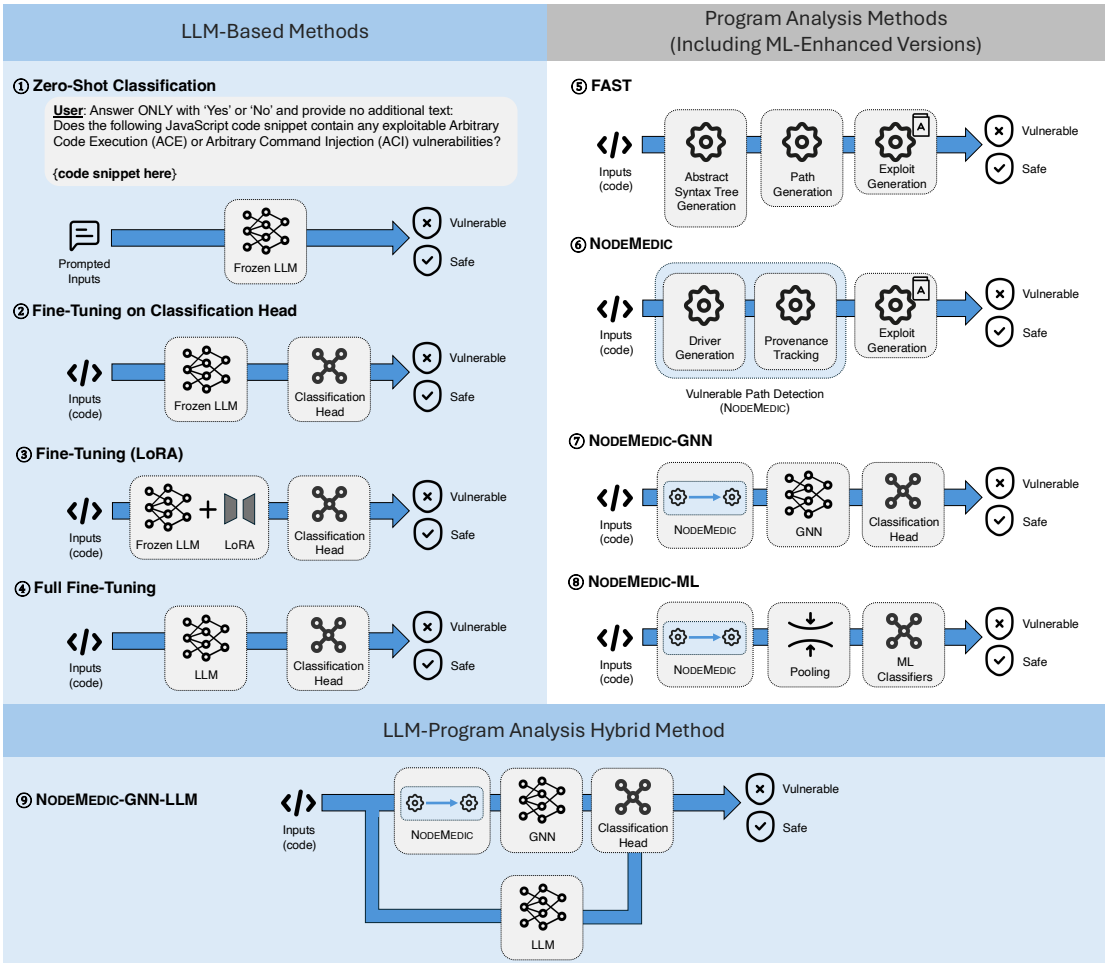

Figure 7.1: Nodemedic-LLM vulnerability detection approach

7.1.2 LLM-based Methods

Common practices in leveraging large language models (LLMs) for vulnerability detection typically fall into two categories:

1. **Zero-shot and few-shot prompting**, where the model is provided with code snippets along with carefully designed natural language queries to identify security risks. This approach benefits from LLMs' generalization ability but often struggles with nuanced vulnerabilities that require deeper program understanding.
2. **Fine-tuning**, where the model is trained on labeled vulnerability datasets to learn domain-specific patterns. Fine-tuning can significantly improve detection accuracy but comes with high computational costs and data collection challenges. Other than full fine-tuning, there are also lightweight fine-tuning methods, such as LoRA fine-tuning [26, 42] and fine-tuning on only selected layers.

To systematically evaluate LLMs in vulnerability detection, we assess several models under different settings, including zero-shot classification, fine-tuning on a classification head, LoRA

fine-tuning [26, 42], and full fine-tuning. We exclude few-shot learning in this work because the large input size of the code snippets makes it difficult to fit multiple samples into a reasonable context window. The code snippet given to LLMs is the file that contains potential sinks. If the file is too long to fit within the predefined context length, we truncate it, taking code immediately around the sink. This is based on our observations that the majority of vulnerable logic is local to the sink and the surrounding code in the dataset.

Zero-Shot Classification (① in Figure 7.1)

In this setting, we use an auto-regressive generation head that enables the LLM to generate a textual response indicating whether a given JavaScript package contains vulnerabilities. Zero-shot classification relies entirely on the LLM's pre-trained knowledge and ability to generate a relevant answer token by token in an auto-regressive manner.

For this evaluation, we prompt the LLM with a common query:

> **User:** Answer ONLY with 'Yes' or 'No' and provide no additional text:
> Does the following JavaScript code snippet contain any exploitable Arbitrary Code Execution (ACE) or Arbitrary Command Injection (ACI) vulnerabilities?
>
> `{code snippet here}`

For models that can run locally, we disable sampling during generation to ensure deterministic results. For some models that require a cloud-based API, the classification responses may vary slightly across multiple runs. Even though we instruct the model to output only "Yes" and "No," for models trained on a Chain-of-Thought (CoT) objective [128], the model may still produce extra text wrapped in special tokens as part of the reasoning process before the final answer. We ignore these additional tokens and only consider the final answer. The outputs are filtered based on the presence of "Yes" which is considered vulnerable, while all other cases, including those that generate neither "Yes" nor "No," are considered non-vulnerable.

Fine-Tuning on a Classification Head (② in Figure 7.1)

Instead of using a generation head, we attach a classification head on top of the base LLMs and fine-tune the model for the vulnerability detection task while keeping the base LLMs frozen (not involved in training). The classification head takes the embedding from the last non-padding position of the output from the last attention layer as input and produces an output shape of 2 logits, representing the two classes (vulnerable or non-vulnerable). We use the cross-entropy loss function to train the classification head with weights that correspond to the class imbalance in the dataset.

LoRA Fine-Tuning (③ in Figure 7.1)

LoRA (Low-Rank Adaptation) fine-tuning [26, 42] offers a lightweight approach to adapting LLMs without updating all parameters. Instead of modifying the entire model, LoRA injects low-

rank adapters into selected layers of the model. These adapters are small, low-rank matrices that are learned during fine-tuning. Similar to the previous method, we attach a classification head on top of the base LLMs and fine-tune the model for the vulnerability detection task. However, in this case, we only update the low-rank adapters and the classification head, while the base LLMs remain frozen. The same cross-entropy loss function is used to train the classification head.

Full Fine-Tuning (④ in Figure 7.1)

In full fine-tuning, we update all parameters of the LLM and the classification head using our labeled vulnerability dataset, applying the same cross-entropy loss function as previously described.

During the fine-tuning of all the aforementioned LLM-based methods, all frozen parameters are stored in 4-bit NormalFloat (NF4) precision for memory efficiency [26], while the trainable parameters are in 16-bit BrainFloat (BF16) precision.

Note that you don't explain 5 (FAST) and 6 (NodeMedic)

7.1.3 ML-enhanced Program Analysis Methods

NODEMEDIC-GNN and NODEMEDIC-ML utilize NODEMEDIC's taint provenance tracking component to create provenance graphs (as described in Section 2.6.3) in both the training and inference pipelines. The operation, tainted status, and sink type of each node are used as inputs. Additionally, the vulnerability type is included as an input for the entire graph. The 100 most common operations in our dataset (as described in Section 3.1) are assigned class numbers `0` to `99`. Class `100` is designated for less frequent operations, while class `101` is used for empty or missing operation attributes in the provenance graphs. Tainted statuses are encoded as class `0` for `False` (untainted), class `1` for `True` (tainted) and class `2` for missing attributes. Sink types are represented with class `0` for `spawn`, class `1` for `exec`, and class `2` for missing attributes. Vulnerability types are encoded as class `0` for ACE and class `1` for ACI vulnerabilities.

Each attribute is represented as a one-hot vector, where the corresponding class has a value of `1`, and all other classes have a value of `0`. The four one-hot vectors are then concatenated to form the embedding for a single node in the graph. Together, the graph's topology and the embeddings of its nodes make up the complete representation of the graph.

NODEMEDIC-GNN (⑦ in Figure 7.1)

The GNN component in NODEMEDIC-GNN starts with a Gated Graph Sequence Neural Network (GGNN) [70], which is a specialized type of neural network designed to learn from graph-structured data by capturing dependencies and relationships between nodes. The GGNN works by iteratively passing messages along edges, enabling each node to gather information from its neighbors and update its representation based on the graph's structure and features. In the final step, the learned abstract node embeddings are combined into a graph-level representation using Global Attention Pooling [70], resulting in the final graph embedding. The graph embedding is then fed into a classification head to predict vulnerability.

NODEMEDIC-ML (⑧ in Figure 7.1)

In NODEMEDIC-ML, only the node embeddings of the graph are taken into account, while the topology is disregarded. The embeddings of all nodes are first fed into a pooling layer to create a unified shape embedding vector that represents the entire graph, regardless of the number of nodes. The pooled embedding is then passed through machine learning classifiers to predict the vulnerability.

7.1.4 LLM-Program Analysis Hybrid Methods

To bridge the gap between program analysis and LLM-based reasoning, we evaluate a hybrid approach (⑨ in Figure 7.1) that combines NODEMEDIC-FINE's vulnerability path detection, graph neural networks (GNNs), and large language models (LLMs). This method, referred to as NodeMedic-GNN-LLM , integrates the strengths of both program analysis and LLMs to enhance vulnerability detection.

This hybrid approach uses the output from the GNN model and the output from the LLM as its embeddings. The two embeddings are concatenated and sent through a classification head to predict the vulnerability. The full model is trained end-to-end, with the GNN and LLM components updated simultaneously.

7.2 Experiment Setup

We first outline the model selection and implementation (Section 7.2.1) used in our evaluation. Next, we discuss the evaluation metrics (Section 7.2.2) and the system configuration (Section 7.2.3).

7.2.1 Model Selection and Implementation

You need to explain what is FAST before

We directly report the latest results from the NODEMEDIC-FINE experiments in Section 7.3. We evaluated FAST [1] against our dataset. For FAST, we enabled the `-X` flag to turn on auto-exploit generation. Additionally, for FAST, we used the `-t` flag to specify vulnerability types as `os_command` and `code_exec`, which correspond to ACI and ACE vulnerabilities, respectively.

For NODEMEDIC-ML, we evaluate logistic regression, support vector machine (SVM), random forest, and XGBoost. These methods are trained on provenance graphs generated by NODEMEDIC-FINE. For logistic regression, SVM, and random forest experiments, we used classes from the `scikit-learn` library, while the `xgboost` package was used for XGBoost experiments. For these machine learning baseline models, default hyperparameters were applied, and the random state was set to 42 for reproducibility.

For NODEMEDIC-GNN, we use the GatedGraphConv and GlobalAttentionPooling modules from the Deep Graph Library [125].

For the LLM-based methods, we experiment with DeepSeek-R1-Distill-Qwen-14B, DeepSeek-R1-Distill-Llama-8B, DeepSeek-R1-Distill-Qwen-7B [24], Llama-3.1-8B-Instruct

[1] FAST implementation accessed at `https://github.com/fast-sp-2023/fast`

[28], Qwen2.5-Coder-14B-Instruct, and Qwen2.5-Coder-7B-Instruct [142]. We utilized the implementation and parameters of the models provided by Hugging Face. Additionally, we experiment with two models evaluated via cloud APIs: OpenAI o3-mini-high and DeepSeek-R1 [24]. The random state is set to 0 to ensure reproducibility. For LoRA, we are using a rank of 256 and an alpha of 512 for all experiments. An ablation study on LoRA's hyperparameters is presented in Section 7.3.3.

7.2.2 Evaluation Metrics

NODEMEDIC-FINE, FAST, and zero-shot LLM models are evaluated directly on the test dataset, while other models that require training are trained on the training dataset and validated on the validation dataset during the training process. The best model from training is then evaluated on the test dataset. To assess the performance of each method, we employ the following metrics: $F1 = \frac{TP}{TP+0.5(FP+FN)}$, $Precision = \frac{TP}{(TP+FP)}$, $Recall = \frac{TP}{TP+FN}$ (where TN is true negative, TP is true positive, FP is false positive, and FN is false negative). These are the same metrics used for prior work on vulnerability detection [34, 67, 113, 141].

7.2.3 System Configuration

Evaluations that require only CPUs are conducted on a computing cluster. Each task runs individually in a virtually isolated environment with a 2-core CPU, which is part of an AMD EPYC 7742 processor, and 16 GB of RAM. The timeout for each task is set to 36 hours. For experiments that require GPUs, except for those where LLMs are undergoing full fine-tuning, each task is executed in a virtual environment with one NVIDIA H100 (80GB) GPU, two Intel Xeon 8480C PCIe Gen5 CPUs (each with 56 cores running at 2.0/3.8 GHz), and 2 TB of RAM. For LLMs that require full fine-tuning, we use a computing cluster with four NVIDIA H100 (80GB) GPUs, two Intel Xeon 8480C PCIe Gen5 CPUs (each with 56 cores running at 2.0/3.8 GHz), and 2 TB of RAM.

7.3 Results

Our experiments aim to address the following research questions:

RQ1: Effectiveness of LLMs. How effective are LLM-based methods in detecting exploitable vulnerabilities in Node.js packages compared to traditional program analysis approaches and ML-enhanced program analysis methods?

RQ2: Cross-Method Prediction Comparison. How do the predictions of graph-based models (e.g., GNN-enhanced NODEMEDIC-FINE) and LLM-based methods differ, and does their combination lead to improved vulnerability detection?

RQ3: Ablation Study on LLMs. How do different large language models and usage strategies (e.g., zero-shot, various fine-tuning methods) perform in our vulnerability detection task?

RQ4: Training and Inference Time. How do LLM-based methods compare to traditional program analysis approaches in terms of prediction latency and pre-prediction (training) time?

Table 7.2: F1 Score (F1), Precision (Prec), Recall (Rec), and Accuracy (Acc) for all program analysis-based methods and LLM-based methods. For LLMs, methods are categorized with the base LLMs. Higher metrics indicate better performance. - indicates that the calculation of such metrics is undefined due to division by zero. *Bold* indicates the best F1 score within each group; underline indicates the best F1 score overall.

Model	F1	Prec	Rec	Acc
Random ($P_{vuln} = 1/2$)	0.596	0.738	0.500	0.500
Random ($P_{vuln} = 1173/1640$)	0.727	0.738	0.715	0.602
Random ($P_{vuln} = 1$)	**0.849**	0.738	1.000	0.738
Random ($P_{vuln} = 0$)	-	-	0.000	0.262
FAST	0.699	0.915	0.566	0.641
NODEMEDIC-FINE				
–w/ auto confirmation	0.744	1.000	0.592	0.699
–w/ GNN	**0.943**	0.955	0.932	0.917
–w/ Random Forest	0.929	0.911	0.947	0.893
–w/ XGBoost	0.931	0.928	0.934	0.898
–w/ Logistic Regression	0.917	0.889	0.947	0.874
–w/ SVM	0.914	0.883	0.947	0.869
OpenAI o3-mini-high				
–zero-shot	0.733	0.978	0.586	0.684
DeepSeek-R1				
–zero-shot	0.858	0.842	0.875	0.786
Llama-3.1-8B-Instruct				
–zero-shot	0.844	0.741	0.980	0.733
–cls-head-only-ft	0.843	0.735	0.987	0.728
–lora-ft	0.922	0.876	0.974	0.879
–full-ft	0.849	0.738	1.000	0.738
–full-ft + GNN	0.850	0.761	0.967	0.748
Qwen2.5-Coder-7B-Instruct				
–zero-shot	0.353	0.943	0.217	0.413
–cls-head-only-ft	0.849	0.738	1.000	0.738
–lora-ft	0.900	0.857	0.947	0.845
–full-ft	**0.933**	0.907	0.961	0.898
–full-ft + GNN	0.919	0.888	0.952	0.875

7.3.1 RQ1: Effectiveness of LLMs

To assess the effectiveness of LLM-based methods in detecting exploitable vulnerabilities, we compare them against traditional and ML-enhanced program analysis approaches. The evaluation focuses on F1 score, Precision, Recall, and Accuracy, where higher values indicate better detection performance.

Table 7.2 shows the full results of our evaluations. We include four naive baselines for comparison: Random ($P_{vuln} = 1/2$), Random ($P_{vuln} = 1173/1640$), Random ($P_{vuln} = 1$), and Random ($P_{vuln} = 0$). The first baseline randomly predicts vulnerabilities with a probability of $1/2$,

the second baseline uses the ratio of vulnerable packages in the training split of our dataset, the third baseline always predicts vulnerabilities, and the fourth baseline never predicts vulnerabilities. The performance of methods using GNNs is based on the average of three runs, as the results are not deterministic when using GPUs. In contrast, other methods are deterministic and are evaluated with just one run.

Minor: I think in previou sections you use the bold with a . to denote these paragraphs.

Program Analysis Approaches Traditional program analysis methods, such as FAST and NODEMEDIC-FINE, have high precision, but low recall, indicating that they often miss vulnerabilities. NODEMEDIC-FINE with auto-confirmation has better precision and recall than FAST, because FAST only generates potential exploits, some of which do not work.

ML-enhanced program analysis significantly improves performance: NODEMEDIC-GNN achieves the highest F1 score (0.943) among all methods. Other ML-enhanced methods (e.g., Random Forest, XGBoost, SVM) also perform well, with F1 scores ranging from 0.914 to 0.931. This indicates that combining machine learning with program analysis enhances detection accuracy and recall without significantly sacrificing precision.

LLM-based Methods LLM-based methods exhibit significant variability based on whether they are used in a zero-shot setting or fine-tuned with various strategies. Overall, zero-shot LLMs tend to perform poorly. Even very large commercial models (e.g., OpenAI o3-mini-high) do not perform well in zero-shot settings. Fine-tuned LLMs generally achieve higher F1 scores. However, fine-tuning only the classification head results in notably lower performance compared to LoRA and full fine-tuning. We will provide a more detailed analysis of the results in Section 7.3.3. Besides, adding NODEMEDIC-FINE features and GNN embeddings to LLMs does not always improve performance. For example, Qwen2.5-Coder-7B-Instruct with NODEMEDIC-FINE features and GNN embeddings performs worse than the same model without these features. We will discuss this in more detail in Section 7.3.2.

Overall, ML-enhanced program analysis methods (e.g., NODEMEDIC-GNN, F1: 0.943) outperform both traditional program analysis and even all fine-tuned LLMs. The best fine-tuned LLMs, Qwen2.5-Coder-7B (F1: 0.933), perform similarly to ML-enhanced methods but require significant training resources. Zero-shot LLMs are inconsistent and can sometimes perform worse than random baselines. Traditional program analysis methods struggle with recall, which can limit their practical application in real-world scenarios. However, because they can generate proof-of-concept exploits, they may be more suitable for tasks where PoC or perfect precision is necessary. In contrast, ML-enhanced methods can offer a higher recall and higher F1 score solution. One advantage of LLM-based methods is that they do not require the same level of domain expertise in specific programming languages or security to design features or rules as program analysis methods do, making them easier to design and implement.

RQ1 Summary: GNN-enhanced NODEMEDIC-FINE performs best among all methods, while certain fine-tuned LLMs can achieve similar performance. Traditional program analysis methods struggle with low recall but can provide PoC exploits and perfect precision.

Make these boxes the same style as previous chapters

Table 7.3: Training and inference times for different models and methods. The inference time is measured per sample in a batched setting: calculated by dividing batch inference time by batch size for methods that support batch inference. Model saving and loading times are not included in the time measurement. *Other ML classifiers* refer to Random Forest, XGBoost, Logistic Regression, and SVM.

Model	Computation Time	
	Training	Inference
FAST	0min	31.6s
NODEMEDIC-FINE		
–w/ auto confirmation	0min	0.79s
–w/ GNN	25min	0.79s
–w/ other ML classifiers	22min	0.79s
DeepSeek-R1-Distill-Qwen-14B		
–zero-shot	0min	15.37s
Llama-3.1-8B-Instruct		
–zero-shot	0min	0.48s
–cls-head-only-ft	11mins	0.11s
–lora-ft	34mins	0.11s
–full-ft	24mins	0.10s
Qwen2.5-Coder-7B-Instruct		
–zero-shot	0min	0.25s
–cls-head-only-ft	10mins	0.10s
–lora-ft	30mins	0.10s
–full-ft	20mins	0.06s

7.3.2 RQ2: Cross-Method Prediction Comparison

Large Language Models

We examine how LLMs make predictions based on the input code. For this evaluation, we use DeepSeek-R1-Distill-Qwen-7B as a representative model. We compare the Shapley values of the classification-head-only fine-tuning version with the full fine-tuning to see how the model's focus shifts during training. Specifically, we choose the packages where, in the testing split, the classification-head-only fine-tuning makes incorrect predictions, while the fully fine-tuned version makes correct predictions. The classification head-only fine-tuning does not alter the LLM's core components, preserving most of its pre-trained parameter values. We use the implementation of SHAP (SHapley Additive exPlanations) [81] to generate the Shapley values.

We found that for non-vulnerable packages, the sink function *spawn* consistently exhibited a high Shapley value, strongly contributing to the model's classification as "not vulnerable."

Figure 7.2 is an example of an invulnerable package *third-party-resources-checker*, where the presence of *spawn* is the most significant feature in the prediction after full fine-tuning. This suggests that the presence of *spawn* is associated with safer execution patterns compared to other process creation functions.

```
'use strict';

var fs = require('fs');
var phantomPath = require('phantomjs-prebuilt').path || '/usr/local/bin/phantomjs';
var Promise = require('promise');
var script = fs.realpathSync(__dirname + '/detect-phantom.js');
var spawn = require('child_process').spawn;

exports.check = function check (uri, whitelist) {
    return new Promise(function (resolve, reject) {
        var args = [script, uri];
        if (whitelist) args.push(whitelist);

        var phantomjs = spawn(phantomPath, args);
        var buffer = '';

        phantomjs.stdout.on('data', function(data) { buffer += data; });

        phantomjs.on('exit', function(code){
            var stdout = buffer.split("\n");
            stdout.pop();
            resolve([code, stdout]);
        });
    });
}
```

Figure 7.2: Shapley values of the full-fine-tuned DeepSeek-R1-Distill-Qwen-7B on a package *third-party-resources-checker*. The correct prediction is: ”not vulnerable.” Red values indicate a positive contribution to the vulnerable prediction, while blue values indicate a negative contribution. Darker colors represent a higher absolute Shapley value.

Minor fix “”

One key reason for this is how *spawn* handles its arguments. Unlike *exec*, which interprets a string directly as shellcode, *spawn* takes a command and its arguments separately as elements of an array, and takes a second configuration object that only allows for shell meta-character evaluation if explicitly configured.

This design significantly reduces the risk of command injection attacks because an attacker must control both the command array and the configuration object (unless the package itself takes the unsafe action to allow shellcode evaluation). The need to control multiple arguments was specifically noted by NodeMedic-FINE as a challenge for confirming/dis-confirming exploitability of *spawn*.

Should this be disconfirming ?

For other cases, no specific pattern stands out, and most tokens contribute only marginally to the model’s predictions. This suggests that, apart from certain key indicators like *spawn*, the vulnerability classification relies on a distributed set of features rather than any single dominant

token. The Shapley values for these other tokens tend to be small and dispersed, indicating that their individual influence on the final prediction is limited. This behavior supports the idea that security vulnerabilities often result from complex interactions among different parts of the code, rather than being linked to the presence or absence of a single token.

Graph-Based Models

We compared the predictions of NODEMEDIC-GNN with the fully fine-tuned DeepSeek-R1-Distill-Qwen-7B. Among 206 packages in the test split, 178 packages were predicted correctly by the LLM while 189 packages were predicted correct by NODEMEDIC-GNN. Out of all the predictions on the test split made by LLM and NODEMEDIC-GNN: 171 cases were correctly classified by both classifiers. 10 cases were misclassified by both.

Additionally, we found that NODEMEDIC-GNN and LLMs perform very differently on ACI and ACE vulnerabilities. NODEMEDIC-GNN significantly outperforms LLMs on ACE vulnerabilities, achieving an F1 score of 0.955 compared to 0.889. In contrast, LLMs, along with traditional program analysis methods (i.e., FAST and NODEMEDIC-FINE), perform better on ACI than on ACE. This indicates that the NODEMEDIC-GNN and LLMs may concentrate on different aspects of the package when making predictions. However, combining LLMs and GNNs does not always yield better results than using LLMs alone.

RQ2 Summary: LLMs' identification of non-vulnerable ACI relies on presence of the *spawn* sink, which is typically safer than *exec*. In other cases, no single token determines the prediction. Graph-based methods exhibit a different prediction distribution to LLMs across the test split, but combining LLMs and GNNs does not always improve performance.

Make these boxes the same style as previous chapters

7.3.3 RQ3: Ablation Study on LLMs

Our experiments show that the performance of LLMs in the vulnerability detection task is very sensitive to both the base model and the selected usage strategy (e.g., zero-shot, various fine-tuning methods). Table 7.2 includes key metrics for various LLM-based approaches under different operational configurations, including zero-shot and multiple fine-tuning strategies.

Zero-Shot In a zero-shot setting, OpenAI's o3-mini-high model achieves an F1 score of 0.733, which is lower than five out of the seven other models we evaluated for the zero-shot scenario. However, its metrics—specifically, high precision and moderate recall—are quite similar to those achieved by pure program analysis methods like NODEMEDIC-FINE. This suggests that the model tends to be conservative in its predictions.

Among other zero-shot LLMs, all models show fairly similar results, except for DeepSeek-R1-Distill-Qwen-7B and Qwen2.5-Coder-7B-Instruct, which have significantly lower F1 scores. DeepSeek-R1-Distill-Qwen-7B, in particular, sometimes outputs a final answer that contradicts its prior reasoning. The following are two examples where, during the reasoning, the model believes the code is exploitable, but the final answer is "no," and in the other example, it is the opposite.

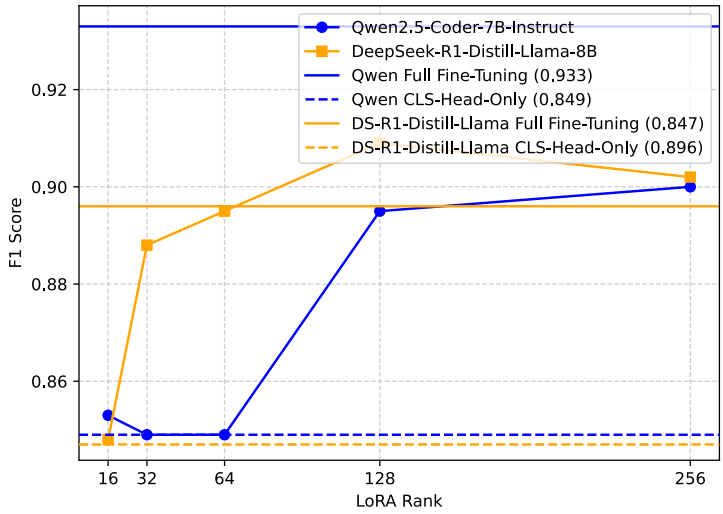

Figure 7.3: Comparison of various LoRA ranks on the F1 score of DeepSeek-R1-Distill-Llama-8B and Qwen2.5-Coder-7B-Instruct. The F1 score is computed based on the test split. Full fine-tuning and classification-head-only fine-tuning are included for comparison.

Output 1:

$< think >$ ... I think **there are** exploitable ACE vulnerabilities here because the code allows for the execution of arbitrary functions based on the args parameter, which could be controlled by an attacker. $< /think >$ **No**

Output 2:

$< think >$ ... I think the code **does not have** any exploitable ACE or ACI vulnerabilities because the unshift method is overridden to do nothing, preventing any code execution. $< /think >$ **Yes**

LoRA versus Full Fine-Tuning It is widely believed that a properly hyper-parameter-tuned Low-Rank Adaptation (LoRA) method can achieve performance similar to or even better than full fine-tuning [26, 42, 83]. We found similar results showing that LoRA fine-tuning performs almost as well as full fine-tuning for most models. For DeepSeek-R1-Distill-Llama-8B and Llama-3.1-8B-Instruct, LoRA fine-tuning even outperforms full fine-tuning. This suggests that LoRA fine-tuning can be a more efficient and effective approach for fine-tuning LLMs, especially when computational resources are limited. Figure 7.3 shows the comparison of various LoRA ranks on the models' F1 scores. A sudden increase in the F1 score is observed when the rank is increased, but additional increases in rank do not significantly enhance performance.

RQ3 Summary: LLM performance in vulnerability detection is highly sensitive to both model choice and usage strategy. In zero-shot settings, most models perform similarly. LoRA fine-tuning can achieve similar performance to full fine-tuning, and in some cases, it can even outperform full fine-tuning.

Make these boxes the same style as previous chapters

7.3.4 RQ4: Training and Inference Time

To compare the computational efficiency of LLM-based methods against traditional and ML-enhanced program analysis approaches, we analyze both training time (pre-prediction) and inference time (prediction latency), as presented in Table 7.3. All full fine-tuning of the LLMs is performed on four (4) GPUs, unlike other methods for LLMs where experiments are conducted using one (1) GPU. For FAST, the inference time is the median overhead reported in the paper [46] that introduces FAST. The graph generation times of NODEMEDIC-FINE represent the average tool runtime reported in the paper [14], which is 0.79 seconds per output. Some methods of DeepSeek-R1-Distill models are not reported because they share the same model structure as the non-distilled models, and their performance is expected to be similar.

Full fine-tuning or LoRA fine-tuning of LLMs usually takes more training time than ML-enhanced program analysis tools. However, the inference time for fine-tuned LLMs is quicker than that of both program analysis methods and ML-enhanced program analysis methods when processing in batches. FAST and NODEMEDIC-FINE provide a PoC exploit, while the other methods cannot.

RQ4 Summary: LLM-based methods have slightly higher training costs compared to ML-enhanced program analysis methods, but they remain within a reasonable range for practical use. Regarding inference time, fine-tuned LLMs are significantly faster (up to 10 times) than both program analysis methods and ML-enhanced program analysis methods, although the cost of hardware resources should be taken into account.

Make these boxes the same style as previous chapters

7.4 Conclusion

Our study demonstrates that combining machine learning with program analysis can enhance the results of vulnerability detections in Node.js packages. Our findings show that applying machine learning to outputs from existing program analysis or vulnerability detection tools, or fine-tuning large language models for this task, can significantly improve the performance of vulnerability detection for real world Node.js packages. Future work could explore the co-design of program analysis tools and machine learning techniques—particularly advanced deep learning models—as auxiliary components to further enhance their capability to identify vulnerabilities more effectively and accurately.

8 Conclusion

In summary, this thesis proposal describes three LLM based tools for software fault localization, patch efficiency, and automatic programming language transpilation. My work is the first to use LLMs for all stages of program repair, as well as the first to propose tooling on combining LLMs with automated verification on the task of program transpilation.

The implementations of the techniques completed in this thesis proposal are open to the publicly and already used by researchers studying related work. For the remainder of our PhD, we propose to expand our fault localization work to detect entire functions as potentially vulnerable across multiple files in a larger repository.

Your conclusion is very short. You should expand this. You can relate the contributions of each section with your thesis section. You can also have a subsection called “Open Challenges and Future Work” where you can write at least 1-page about that topic. In total it would be nice that this section is at least 2 pages.

Bibliography

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing*, pages 39–46. IEEE, 2006. 2.2, 3.2.2

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007. 1, 2.2

[3] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Computing Surveys*, 2017. URL https://doi.org/10.1145/3106739. 2.6.2

[4] Timos Antonopoulos, Eric Koskinen, Ton Chanh Le, Ramana Nagasamudram, David A. Naumann, and Minh Ngo. An algebra of alignment for relational verification. *Proceedings of the ACM on Programming Languages*, 7(POPL):20:573–20:603, January 2023. doi: 10.1145/3571213. URL https://dl.acm.org/doi/10.1145/3571213. 2.4

[5] Simone Balloccu, Patrícia Schmidtová, Mateusz Lango, and Ondřej Dušek. Leak, cheat, repeat: Data contamination and evaluation malpractices in closed-source llms. *arXiv preprint arXiv:2402.03927*, 2024. 4

[6] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994. 3.1.2

[7] Stella Biderman, USVSN PRASHANTH, Lintang Sutawika, Hailey Schoelkopf, Quentin Anthony, Shivanshu Purohit, and Edward Raff. Emergent and predictable memorization in large language models. *Advances in Neural Information Processing Systems*, 36, 2024. 5.2.2

[8] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022. 1, 2.1

[9] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022. 3, 3.1.1

[10] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. *Provably-Safe* multilingual software

sandboxing using *WebAssembly*. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1975–1992, 2022. 2.5, 5.1.2

[11] Peter F Brown, Vincent J Della Pietra, Peter V Desouza, Jennifer C Lai, and Robert L Mercer. Class-based n-gram models of natural language. *Computational linguistics*, 18 (4):467–480, 1992. 4

[12] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023. 1

[13] Darion Cassel, Nuno Sabino, Ruben Martins, and Limin Jia. Nodemedic-fine: Automatic detection and exploit synthesis for node. js vulnerabilities. 2.6.1, 2.6.2

[14] Darion Cassel, Wai Tuck Wong, and Limin Jia. Nodemedic: End-to-end analysis of node. js vulnerabilities with provenance graphs. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 1101–1127. IEEE, 2023. 2.6.1, 2.6.3, 7.3.4

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. 1, 2.1

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. 1, 3, 3.1.1, 3.1.1

[17] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. Sound Loop Superoptimization for Google Native Client. *ACM SIGARCH Computer Architecture News*, 45(1):313–326, April 2017. ISSN 0163-5964. doi: 10.1145/3093337.3037754. URL https://dl.acm.org/doi/10.1145/3093337.3037754. 2.4

[18] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1027–1040. Association for Computing Machinery, June 2019. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314596. URL https://doi.org/10.1145/3314221.3314596. 2.4

[19] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19:7–34, 2001. 5.1.5

[20] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004. doi: 10.1007/978-3-540-24730-2_15. URL https://doi.org/10.1007/978-3-540-24730-2_15. 5.1.5

[21] The MITRE Corporation. CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'), 2020–. https://cwe.mitre.org/data/

definitions/22.html. 2.6.1

[22] The MITRE Corporation. CWE - CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection') (4.3), 2020–. https://cwe.mitre.org/data/definitions/77.html. 2.6.1

[23] The MITRE Corporation. CWE - CWE-94: Improper Control of Generation of Code ('Code Injection') (4.3), 2020–. https://cwe.mitre.org/data/definitions/94.html. 2.6.1

[24] Daya Guo DeepSeek-AI, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. 7.2.1

[25] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pages 345–356, 2016. 2.1

[26] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized LLMs. *Advances in Neural Information Processing Systems*, 36, 2024. 2, 7.1.2, 7.1.2, 7.1.2, 7.3.3

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 2.3

[28] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. 7.2.1

[29] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Translating C to safer Rust. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29, 2021. 2.5, 5.2.2

[30] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020. 6.2.1

[31] George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, 1997. 5.1.5

[32] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022. 5.2.1

[33] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis, 2022. URL https://arxiv.org/abs/2204.05999. 2.1, 4.2.1

[34] Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining*

Software Repositories, pages 608–620, 2022. 6.2.1, 7.2.2

[35] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020. 1

[36] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. AFFOGATO: Runtime detection of injection attacks for Node.js. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018. 2.6.2

[37] Ali Ghanbari and Andrian Marcus. Patch correctness assessment in automated program repair based on the impact of patches on production and test code. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 654–665, 2022. 2.3, 4, 4.2.2, 4.2.4

[38] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*, pages 596–607, 2022. 2.6, 1

[39] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016. 4

[40] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 518–529, 2020. 2.3

[41] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022. 3.2.6

[42] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. 2, 7.1.2, 7.1.2, 7.3.3

[43] Zhiqiang Hu, Yihuai Lan, Lei Wang, Wanyu Xu, Ee-Peng Lim, Roy Ka-Wei Lee, Lidong Bing, and Soujanya Poria. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models. *arXiv preprint arXiv:2304.01933*, 2023. 5.2.1

[44] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. *arXiv preprint arXiv:2303.07263*, 2023. 1

[45] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014. 3, 3.2.1, 4.2.2

[46] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, VN Venkatakrishnan, and Yinzhi Cao. Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1059–1076. IEEE, 2023. 7.3.4

[47] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. 1

[48] R. Karim, F. Tip, A. Sochurkova, and K. Sen. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering*, 2018. 2.6.2

[49] Misoo Kim, Youngkyoung Kim, Kicheol Kim, and Eunseok Lee. Multi-objective optimization-based bug-fixing template mining for automated program repair. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5, 2022. 4

[50] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE symposium on security and privacy (SP)*, pages 595–614. IEEE, 2017. 2.6, 1

[51] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. The stack: 3 TB of permissively licensed source code. *arXiv preprint arXiv:2211.15533*, 2022. 5.3

[52] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 165–176, 2016. 3.2.4

[53] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022. 6.2.5

[54] Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. Patch generation with language models: Feasibility and scaling behavior. In *Deep Learning for Code Workshop*, 2022. URL https://openreview.net/forum?id=rHlzJh_b1-5. 2.1

[55] Steve Kommrusch, Martin Monperrus, and Louis-Noël Pouchet. Self-supervised learning to prove equivalence between straight-line programs via rewrite rules. *IEEE Transactions on Software Engineering*, 49(7):3771–3792, July 2023. ISSN 1939-3520. doi: 10.1109/TSE.2023.3271065. 2.4

[56] Charles Koutcheme, Sami Sarsa, Juho Leinonen, Arto Hellas, and Paul Denny. Automated program repair using generative models for code infilling. In *International Conference on Artificial Intelligence in Education*, pages 798–803. Springer, 2023. 1

[57] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014. 2.3

[58] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011. 2.3

[59] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019. 1, 4

[60] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov,

Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023. 4.2.1

[61] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023. 5.2.1, 5.2.2

[62] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. *Detecting Node.Js Prototype Pollution Vulnerabilities via Object Lookup Analysis*. 2021. 2.6.2

[63] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 169–180, 2019. 2.2

[64] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 169–180, 2019. 1, 3.2.2, 3.2.3, 3.2.4, 3.2.7

[65] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. doi: 10.1145/3360588. URL `https://doi.org/10.1145/3360588`. 3.2.1

[66] Yi Li, Shaohua Wang, and Tien Nguyen. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*, pages 661–673. IEEE, 2021. 3.2.2, 3.2.3

[67] Yi Li, Shaohua Wang, and Tien N Nguyen. 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*, pages 292–303, 2021. 2.6, 6.2.1, 6.2, 7.2.2

[68] Yi Li, Shaohua Wang, and Tien N Nguyen. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering*, pages 511–523, 2022. 2.2, 4.1.2

[69] Yi Li, Shaohua Wang, and Tien N Nguyen. Fault localization to detect co-change fixing locations. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 659–671, 2022. 1, 3.2.4

[70] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015. 7.1.3

[71] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd annual conference on computer security applications*, pages 201–213, 2016. 2.6

[72] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection.

arXiv preprint arXiv:1801.01681, 2018. 6.2.1

[73] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R Cordy, and Ahmed E Hassan. In Rust we trust: a transpiler from unsafe C to safer Rust. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 354–355, 2022. 2.5

[74] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35:1950–1965, 2022. 5.2.1, 6.1

[75] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 31–42, 2019. 1, 4, 4.1.2

[76] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 615–627, 2020. 4, 4.2.3, 4.2.4

[77] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019. 6.2.1

[78] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312, 2016. 1

[79] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 664–676, 2021. 2.2

[80] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 664–676, 2021. 1, 3.2.1, 3.2.3, 3.2.4, 3.2.7

[81] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf. 7.3.2

[82] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele,

Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021. doi: 10.1109/TSE.2019.2946563. 2.6

[83] Yuren Mao, Yuhang Ge, Yijiang Fan, Wenyi Xu, Yu Mi, Zhonghao Hu, and Yunjun Gao. A survey on lora of large language models. *Frontiers of Computer Science*, 19(7):197605, 2025. 7.3.3

[84] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE)*, pages 691–701. ACM, 2016. 4

[85] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1169–1180, 2022. 2.2, 4.1.2

[86] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1169–1180, 2022. 3.2.1, 3.2.2, 3.2.3

[87] Marcus J Min, Yangruibo Ding, Luca Buratti, Saurabh Pujar, Gail Kaiser, Suman Jana, and Baishakhi Ray. Beyond accuracy: Evaluating self-consistency of code llms. In *The Twelfth International Conference on Learning Representations*, 2023. 1

[88] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined symbolic taint analysis. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association. 2.6.2

[89] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. {VulChecker}: Graph-based vulnerability localization in source code. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6557–6574, 2023. 1

[90] Mozilla. JavaScript eval function documentation, 2022. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval`. 2.6.1

[91] Mozilla. JavaScript Function constructor documentation, 2022. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function`. 2.6.1

[92] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540, 2007. 2.6

[93] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: Feedback-driven static analysis of Node.js applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019. 2.6.2

[94] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022. 3, 3.1.1

[95] Node.js. Node.js exec API documentation, 2022. https://nodejs.org/api/child_process.html#child_processexeccommand-options-callback. 2.6.1, 2.6.1

[96] Node.js. Node.js execSync API documentation, 2022. https://nodejs.org/api/child_process.html#child_processexecsynccommand-options. 2.6.1

[97] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. 1

[98] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209, 2011. 3.2.4

[99] Nishant Patnaik and Sarathi Sahoo. Javascript static security analysis made easy with JSPrime. In *Blackhat USA*, 2013. 2.6.2

[100] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. Does genetic programming work well on automated program repair? In *2013 International Conference on Computational and Information Sciences*, pages 1875–1878. IEEE, 2013. 2.3

[101] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015. 2.3

[102] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the" naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439, 2016. 2.1, 4

[103] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the" naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439, 2016. 3.2.1

[104] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, pages 419–428, 2014. 2.1

[105] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33, 2020. 1, 5.2.3

[106] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023. 5.2.2, 6.1.2, 6.2.1

[107] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir,

Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018. 6.2.1

[108] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1): 61–80, 2008. 2.2

[109] Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surv.*, 44(3), jun 2012. ISSN 0360-0300. doi: 10.1145/2187671.2187673. URL https://doi.org/10.1145/2187671.2187673. 2.6

[110] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 532–543, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786825. URL https://doi.org/10.1145/2786805.2786825. 2.3, 4

[111] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel. Extracting Taint Specifications for JavaScript Libraries. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020. 2.6.2

[112] Cristian-Alexandru Staicu, M. Pradel, and B. Livshits. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS. In *NDSS*, 2018. 2.6.1

[113] Benjamin Steenhoek, Hongyang Gao, and Wei Le. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024. 2.6, 6.2.1, 6.2, 7.2.2

[114] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578*, 2022. 5.2.2, 5.2.3

[115] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013. doi: 10.1109/SP.2013.13. 2.6

[116] Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F Bissyandé. The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–34, 2023. 2.3, 4, 4.2.2, 4.2.4

[117] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023. 2.1, 4.2.1

[118] Lewis Tunstall, Leandro Von Werra, and Thomas Wolf. *Natural language processing with*

transformers. O'Reilly Media, Inc., 2022. 3, 3.1.1

[119] Unknown. C2rust. https://c2rust.com/, 2024. 2.5, 5.2.2

[120] Unknown. bindgen. https://github.com/rust-lang/rust-bindgen, 2024. 2.5

[121] Unknown. citrus. https://gitlab.com/citrus-rs/citrus, 2024. 2.5

[122] Unknown. claude. https://www.anthropic.com/index/introducing-claude, 2024. 5.2.2

[123] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying dynamic trait objects in Rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 321–330, 2022. 5.1.5

[124] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. 3.1.1, 3.1.2

[125] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019. 7.2.1

[126] Qi Wang, Yue Ma, Kun Zhao, and Yingjie Tian. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, 9(2):187–212, 2022. 6.3

[127] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021. 6.2.1

[128] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022. 7.1.2

[129] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 356–366. IEEE, 2013. 2.3

[130] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 1556–1560, 2020. 3.2.1, 3.2.7

[131] Robert F Woolson. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials*, pages 1–3, 2007. 3.2.7, 3.2.7

[132] Dongrui Wu and Jerry M Mendel. Patch learning. *IEEE Transactions on Fuzzy Systems*, 28(9):1996–2008, 2019. 1

[133] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair

in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023. 1

[134] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE, 2023. 2.1, 4, 4.1.2

[135] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th international conference on software engineering*, pages 789–799, 2018. 1, 2.3, 4, 4.2.4

[136] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817*, 2024. 1

[137] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510, 2013. 2.6

[138] Aidan ZH Yang, Ruben Martins, Claire Le Goues, and Vincent J Hellendoorn. Large language models for test-free fault localization. *arXiv preprint arXiv:2310.01726*, 2023. 1, 6.1

[139] Aidan ZH Yang, Sophia Kolak, Vincent J Hellendoorn, Ruben Martins, and Claire Le Goues. Revisiting unnaturalness for automated program repair in the era of large language models. *arXiv preprint arXiv:2404.15236*, 2024. 1

[140] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024. 5.2.1

[141] Aidan ZH Yang, Haoye Tian, He Ye, Ruben Martins, and Claire Le Goues. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. *arXiv preprint arXiv:2406.05892*, 2024. 7.2.2

[142] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024. 7.2.1

[143] Jun Yang, Yuehan Wang, Yiling Lou, Ming Wen, and Lingming Zhang. A large-scale empirical review of patch correctness checking approaches. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1203–1215, 2023. 2.3, 4

[144] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. Automated classification of overfitting patches with statically extracted code features. *IEEE Transac-*

tions on Software Engineering, 48(8):2920–2938, 2021. 2.3

[145] Imam Nur Bani Yusuf and Lingxiao Jiang. Your instructions are not always helpful: Assessing the efficacy of instruction fine-tuning for software vulnerability detection. *arXiv preprint arXiv:2401.07466*, 2024. 6.2.5

[146] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. Ownership guided C to Rust translation. In *Computer Aided Verification (CAV)*, volume 13966 of *LNCS*, pages 459–482. Springer, 2023. 2.5, 5.2.2

[147] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019. 2.6, 3.2.1, 3.2.7