Evaluating LLM Reasoning Strategies and Traditional SAST, A Comparative Study Using the PALADIN Framework

Daniel Hoang
dh3g23@soton.ac.uk
University of Southampton

Brandon Wong Ze Rong
bzrw1n22@soton.ac.uk
University of Southampton

Jonathan Renardo Njoman
jrn1g22@soton.ac.uk
University of Southampton

Eren Rafet
er10g23@soton.ac.uk
University of Southampton

Abstract

The rapid evolution of web and cloud technology has reshaped how applications are built, deployed and maintained. That progress has come with a growing security cost as these systems grow more complex and interconnected. Our research evaluates whether large language models (LLMs) can support vulnerability analysis beyond simple classification by producing outputs that are correct, as well as actionable. PALADIN (Programmatic Analysis and LLM Assessment for Defence Intelligence) is introduced as a structured evaluation framework that decomposes an LLM-generated vulnerability report into security-relevant components and scores each dimension, including labelling, localisation of the vulnerable region, and the quality of explanations, exploit narratives, impact assessments, and mitigations.

A teacher-student annotation pipeline using open-weight Qwen models is implemented, and zero/few-shot prompting is compared against parameter efficient fine-tuning (PEFT) with QLoRA. Experiments use two curated datasets,TitanVul for training and testing, and LLMVul for evaluation-only generalisation. Results show strong performance on simple, pattern-based vulnerabilities but weaker semantic reasoning for complex flaws; fine-tuning improves classification accuracy by ~10% on TitanVul, but degrades localisation, indicating a persistent semantic ceiling in impact and mitigation quality.

1 Introduction

Modern software rarely fails in obvious ways. A single overlooked input check, a subtle logic mistake, or a single missed edge case could be an entry point to data exposure and major system compromise. It is for this reason that thoroughness is increasingly crucial. As systems grow more complex, the probability of exploitable vulnerabilities existing consequently rises [1] [3]. To mitigate these issues, researchers have proposed methods for automatic detection: ranging from static analysis tools, deep learning approaches, and most recently, LLMs.

Despite widespread adoption, existing automated approaches have glaring weaknesses. Traditional Static Analysis Security Tools (SAST) often struggle with contextual reasoning and are generally limited to pattern matching capabilities [3][12]. AI-based scanners, on the other hand, tend to be data-hungry and are often too reliant on predefined vulnerabilities, failing to generalise to other datasets. Furthermore, their opaque structure leads to interpretability issues [18] [2].

Recent studies have assesed large language models (LLMs) such as ChatGPT, Qwen and CodeLlama on their ability to detect software vulnerabilities. Findings suggest that they do, especially when compared with traditional SAST tools and deep learning (DL)-based approaches [13]. For example, it is observed that LLMs perform better when provided with additional context and knowledge, even in simple scenarios, such as Few-shot Prompting. Furthermore, Parameter Efficient Fine-Tuning (PEFT) emerged as a viable approach in enhancing LLMs vulnerability-detection capabilities, allowing for efficient adaptation to tasks under limited computational resources. However, despite these advancements, there is a significant gap in the literature that explore the reasoning and explainability of LLMs when detecting vulnerabilities. Most existing studies either use LLM-as-a-Judge or rely on manual verification, both of which present issues in terms of reliability and scalability. Additionally, there is no standardised framework for automatic evaluation of LLM-generated explanations. This leaves a major challenge in measuring the accuracy of generated explanations are. Much of the LLMs' limitations come in the form of a lack of annotated datasets, data contamination and failing to generalise to imbalanced datasets, which will be explored in greater depth within the literature review.

Research questions

RQ1 How effectively can LLMs detect web and cloud vulnerabilities using prompt-based and PEFT methods?

RQ2 How well can LLMs generate accurate, complete, and actionable explanations on detected vulnerabilities?

Overview of the results Our evaluation through the **P**rogrammatic **A**nalysis & **L**LM **A**ssessment for **D**efence **In**telligence (PALADIN) framework demonstrates that LLMs are highly effective at identifying simple, pattern-based vulnerabilities, yet struggle with the deep semantic reasoning required for complex security flaws.

Addressing **RQ1**, we find that specialised fine-tuning (PEFT) consistently improves classification accuracy by ~10% yet compromises on "localisation intelligence", making the model better at labelling but worse at pinpointing exact vulnerable code.

Addressing **RQ2**, the results highlight a persistent "semantic ceiling"; models fail to accurately assess the broader impact or provide meaningful mitigation strategies. While the LLM shows promise for labelling, its reliance on pattern recognition rather than causal understanding strongly limits its current reliability in a high-stakes environment such as vulnerability detection.

Contributions This study contributes to this field by introducing a novel evaluation framework (PALADIN) that evaluates a model's ability to detect, correctly label, and localise the vulnerability, and provide a mitigation strategy to a given piece of code. In addition, we present the design of our vulnerability report generation pipeline and release the accompanying curated, annotated datasets.

Novelty statement In contrast to prior benchmarks that primarily focus on classification/detection, PALADIN provides a standardised unified method for evaluating not only the prediction accuracy, but also completeness and actionability of a model's reasoning and remediation. Additionally, we introduce an annotation and report-generation pipeline that populates fields such as `sink_or_region` and `exploit_path` which are not typically available in existing datasets.

2 Related Work

2.1 Pre-LLM Vulnerability Detection Systems and the Limits of their Evaluability

Static analysis of source code is a foundational technique in identifying security vulnerabilities in web and cloud-based systems [3]. However, while these methods can be effective for detecting well-understood vulnerability classes, they have significant weaknesses. Becaause they rely on predefined rules, they struggle to detect more complex vulnerabilities that would require a deeper understanding of context within the code. Additionally, the reliance on handcrafted rules makes them difficult to scale across the multitudes of programming languages and evolving software ecosystems [7]. Most of these modern web and cloud systems are characterised by a significant amount of heterogeneity in programming languages. For example, low-level languages such as `C` and `C++` are prone to memory safety errors, whereas higher-level managed and scripting languages often have input validation, injection, or application logic related vulnerabilities. Research indicates that static analysis tools often fail to generalise these concepts, as language-specific semantics and behaviors limit the effectiveness of generic rule sets, leading to high false positive rates, or failing to detect context-dependent vulnerabilities [18] [3].

This limitation is empirically outlined through the study in [7], which evaluated the effectiveness of three widely used, commercial SAST tools across multiple Common Weakness Enumeration (CWE) classes. There were two key issues identified: first, the limited ability of SAST tools to convincingly detect vulnerabilities across CWE types; and second, what they describe as the "price of false positives". It was highlighted that, in practical industrial settings, the high number of false positives would impose a major manual inspection burden on developers. Moreover, due to the lack of explanatory data provided by these tools, the process would not necessarily yield intended the accuracy either, with the act of manual inspection being reported as no more effective than "random guessing". These findings highlight that without actionable reasoning or context, vulnerability reports fail to meaningfully support secure software development.

To address the rigidity and scalability issues of traditional static analysis, research has increasingly explored machine learning and deep learning techniques for automated vulnerability detection. This has the significant advantage of learning representations of code based on data, and not needing to rely on predefined rules [18][16]. Russel et al. [22] show that deep representation learning could be used to detect vulnerabilities across large codebases, reducing reliance on handcrafted rules. Despite these improvements however, deep learning based vulnerability detection systems have several fundamental limitations. Primarily, they are largely dependent on large, labelled datasets. These are expensive to curate and often biased towards a specific label within the dataset, as well as having concerns about overall coverage. Additionally, as noted by Lin [18], models trained on one dataset often fail to generalise to yet unseen projects, leading to them being ineffective in real-world situations.

Furthermore, interpretability remains a major challenge. While neural models may achieve improved detection accuracy, they typically provide little insight into why a piece of code is classified as vulnerable, complicating practical adoption, particularly in security-critical contexts where understanding root causes is essential. Lastly, deep learning models often struggle to capture the broader semantic and contextual reasoning required to understand complex security flaws. Russel [22] and Lin [18] both note that deep learning models tend to excel at recognising recurring patterns but are less effective at reasoning about intent, control-flow semantics, or higher-level security invariants.

2.2 LLM-based vulnerability detection: fine-tuning and prompting approaches

Large Language Models (LLMs) have emerged as a successor to previous approaches as they demonstrate strong effectiveness across natural language and software engineering tasks. Zhou et al. [29] mention that research in the field of LLMs for vulnerability detection and repair has steadily increased, indicating that the cybersecurity community is welcoming of their usage. This growth stems from LLMs' ability to learn features automatically from known vulnerabilities to generalise, find, and fix unseen ones. Moreover, LLMs can be further trained and fine-tuned to enhance their performance, as highlighted by Shestov [23], who noted that LLMs have far surpassed traditional code-specific transformer models (e.g. CodeBERT) and can achieve better results on code understanding tasks which provides an opportunity for improved transfer learning.

Multiple studies have utilised prompting strategies to explore how effective LLMs are in detecting vulnerabilities. Khare et al. [13] evaluated sixteen LLMs of different scales across five vulnerability datasets in `C`/`C++` and `Java` covering twenty-five CWE classes, comparing multiple prompt types of basic, CWE-specific, dataflow-based, few-shot, and Chain-of-Thought (CoT). It was found that performance does not go hand in hand with the scale of an LLM, as smaller models were able to outperform larger ones. On average, LLMs achieved an accuracy of 62.8% and an F1-score of 0.71, performing 10.5% better on synthetic datasets than real-world datasets. While prompting strategies achieved similar accuracies, it is observed that using a CWE-specific prompt yielded a slightly higher accuracy than basic prompts, which suggests additional context improves a model's ability to reason. In comparison to traditional

methods, the static scanner tool CodeQL achieved a higher accuracy but underperformed on certain CWE types. DL models were able to reach similar accuracies but lacked generalizability across datasets and achieved lower F1-scores, with the LLM achieving 0.65 (0.42 and 0.04 higher). Khare et al. [13] concluded that LLMs are preferred as they provide natural language explanations and have low inference costs. Building upon this, Zhou et al. [30] utilised GPT 3.5 to explore combinations of role definition, task description, project information and extra CWE types knowledge within prompts. They found that the performance of the model is highly dependent on the prompt design, as the model showed no practical detection ability (accuracy = 50%, recall = 0%) using the baseline prompt. By integrating external security knowledge such as the 25 most critical CWE types, accuracy improves by 18%. Their best configuration of (base prompt + training samples) yielded an accuracy of 62.7% and a F0.5 score of 62.8%, which outperformed the fully fine-tuned CodeBERT model in accuracy, precision, and F0.5, but underperformed on recall and F1. Preliminary tests using half the test set with GPT-4 showed better results than both GPT3.5 and CodeBERT but came with a higher computational cost. Together, these findings highlight the importance of well-engineered prompts as they can make general LLMs competitive with specialised fine-tuned models for vulnerability detection, but remain limited in recall and cost efficiency.

Parameter Efficient Fine Tuning (PEFT) emerged as an approach to adapt LLMs for domain-specific tasks, achieved through modifying only a small subset of an LLM's parameters while "freezing" the remainder [9]. The LLMVulEXP framework pipeline was proposed by Mao et al. [19] utilising two LLMs in a teacher-student model to evaluate LLM detection ability and explanation quality. They apply LoRA fine-tuning and incorporated CoT prompting during inference. Their methodology led to significant performance gains with LLMVulExp achieving an F1 score of 92.4% compared to 46% for the base CodeLlama model on the SeVC dataset. While comparing with other fine tuned baselines of CodeT5 and CodeBERT, LLMVulExp achieves better F1 score (70.8%) on multi-type detection, demonstrating its ability to detect more vulnerabilities. DiverseVul, on the other hand, achieved an F1 score of 98.8 (34% higher than any other baselines) and remained the top performing model for multi-label CWE detection, showing the fine-tuned model's robustness. However, when evaluated on the real-world PrimeVul dataset, all evaluated models' performance dropped significantly. LLMVulExp's scored 20.7% for macro F1 score of 20.7% on multi-class CWE detection and 39.8% on binary vulnerability detection, highlighting the challenge of generalising to imbalanced real-world datasets. Similarly, Shestov et al. [23] used LoRA fine tuning on WizardCoder and compared it with CodeBERT and ContraBERT on balanced and imbalanced datasets. They found that WizardCoder outperformed ContraBERT slightly (F1 = 0.71 vs 0.68) on balanced datasets, however, this could be attributed to the model size rather than fine tuning. On imbalanced datasets, performance dropped across all models with WizardCoder, achieving an F1 score of 0.27 compared to CodeBERT's 0.22. Both studies conclude that PEFT-based fine-tuning using LoRA has potential to significantly enhance both detection accuracy and explanatory quality in Mao's [19] case.

2.3 Evaluating LLM vulnerability reasoning and explanation quality

Recent studies showed a shift in focus from treating LLMs as classification models toward exploring why these models behave as they do and how they perform reasoning about vulnerabilities. Many LLM-based vulnerability scanners focus on detection or code correction, which can be easily evaluated against a dataset with predefined distinct vulnerability tags and corrected code. However, assessing how well a fine-tuned model explains detected vulnerabilities within code requires more complicated metrics that capture both semantic accuracy and explanatory completeness. If a dataset that has vulnerability and explanation pairs is being used, an LLM's output can be compared against reference explanations from the testing set to evaluate alignment. Conventional similarity metrics are used against the model's outputs to test references using text-overlap (BLEU, ROUGE) [17, 21] and semantic similarity (BERTScore) [27] as well as embedding cosine similarity with Sentence-BERT. These signals, however, do not guarantee correctness in security reasoning; recent approaches incorporate domain-aware evaluation, comparing whether the model's reasoning identifies the same code regions or CWE class [20] as the ground truth. LLM4Vuln [24] provides a unified evaluation framework that separates and assesses LLMs' vulnerability reasoning capabilities. It measured accuracy by using the aforementioned techniques, but also whether the model was able to correctly describe the root cause, exploit mechanism, and mitigation. They found that high-text similarity did not always imply correct security reasoning, which highlights the need for metrics that combine linguistic and semantic evaluation alongside human review for a subset of samples.

Many studies highlight similar limitations in LLM-based vulnerability detection. First, a lack of high-quality datasets. Many existing datasets suffer from noisy or incorrect labels, as they are usually collected in large volumes through an automatic process, validating label correctness is difficult. While manual verification can improve the quality of the dataset, it is a tedious and expensive process especially when the size of dataset scales. Mao et al. [19] emphasizes that most large scale datasets are not annotated with explanations, which limits the ability to evaluate LLMs' reasoning and explanation. To address this, they employ LLMs to perform automatic annotation. However, this introduces a new issue of over-reliance on LLMs, as well as concerns about their reliability in generating accurate explanations. Second, data contamination poses a significant risk during the evaluation process. As most LLMs are pre-trained on open-source code corpora, there is a possibility that the samples from the evaluation dataset overlaps with the training data. In these cases, the LLM would seem to perform well, as it is memorising seen examples rather than generalising to unseen vulnerabilities. To mitigate this issue, evaluation datasets would need to be constructed carefully to ensure that it does not overlap with training data. Thirdly, the complexity of vulnerability data adds another challenge for LLM-based detection. Studies show that LLMs can easily pick up on intra-procedural vulnerabilities with high accuracies, but performance drops significantly when analysing interprocedural vulnerabilities. This is evident in real-world datasets where they are imbalanced, LLMs struggle to generalise. However, there is no standardised method to address

this. Shestov et al. [23] suggests that new strategies would need to be explored to address this issue. Lastly, the wide range and variety of CWE types presents an issue as LLMs tend to perform well on frequently occurring types but struggle to generalise to rarer cases, which highlights the need for balanced training datasets.

3 Methods

3.1 Threat Model

In this section, we outline the threat model of our system, which highlights what could go wrong with our system, who could exploit these weaknesses and what assumptions our system relies on.

Attacker Types Primary attacker: a malicious code submitter operating at inference time. The attacker would target model inference, vulnerability classification and explanation generation, with the goal of evading detection and triggering an incorrect CWE classification, leading to faulty mitigation strategies. The attacker could theoretically achieve this by submitting arbitrary code samples, as well as by providing inputs with altered structures that do not conform to the format expected by the model.

Second attacker: adversarial prompt engineer operating at inference time. The attacker would target the prompt structure, output schema compliance, and explanation quality, with the intention of reducing explanation correctness and manipulating exploit or mitigation outputs. An attacker may exploit this by submitting crafted code to exploit LLM biases, as well as by triggering hallucinations.

Lastly, we could encounter attacks targeting the system infrastructure and/or datasets. In this scenario, an attacker would operate at training time and attempt to disrupt the annotation generation pipeline or training content. For this case, we consider the compromise of infrastructure or datasets as out of scope, aiming to address these cases through system-level assumptions. The first two attackers, however, are considered in scope, as they directly interact with the deployed system.

Attacker Goals As summarised from the previous section, the primary objectives of an attacker are to evade vulnerability detection, induce incorrect CWE attribution or flawed vulnerability reasoning, generate unsafe or incorrect mitigation suggestions, and ultimately reduce the overall reliability of the system.

Attacker Capabilities and Limitations We assume here that the attackers only have access to the deployed model at inference time, without access to training data, model weights, or system infrastructure. Under this assumption, attackers possess the following capabilities:

- Full control over the submitted code logic and intent, allowing them to submit vulnerable or ambiguous code designed to blur CWE boundaries or avoid detection.
- Control over code structure and layout, including the use of unusual control flow, deep nesting, or mixed vulnerability patterns that may affect the quality of sink localisation, exploit paths, or mitigations.
- Control over input size and formatting, such as long functions or inputs near the context window limit.

In contrast, the following limitations apply to the attackers:

- No access to raw datasets or annotation pipelines. Attackers cannot inspect, modify, or influence training data or synthetic annotations used during model development.
- No control over fine-tuning config or training process. Attackers have no ability to alter LoRA/QLoRA configuration, hyperparameters, or sampling strategy.
- No access to model internals during inference. Attacker interacts with the system strictly as a black box, which prevents white-box attacks.
- Limited influence over output structure. While attackers may attempt to submit malformed code and have an influence over output semantics, outputs are fixed to a predefined 7-field JSON schema. This means that any deviations from the expected structure are treated as invalid, preventing field manipulation attacks.

Key Threats to the System Based on the attacker goals and capabilities outlined above, the following can be classified as relevant threats to our system:

- Incorrect CWE classification, where vulnerabilities are detected but attributed to the wrong vulnerability class.
- Hallucinated or unreasonable exploit paths that do not correspond to the actual vulnerable piece of code.
- Unsafe or misleading mitigation suggestions that could lead to new vulnerabilities if applied automatically.
- Incorrect sink or region localisation, essentially invalidating the patched output.

System Assumptions The system we propose operates under the following assumptions:

- Training datasets and synthetic annotations are curated and maintained offline in a trusted environment.
- Fine-tuning and evaluation pipelines are executed in regulated infrastructure and are not subject to external interactions.
- Prompt templates and JSON schema constraints are put in place at deployment time and consistently enforced during inference.
- The underlying hardware, runtime environment, and storage systems are assumed to be secure and uncompromised, as only secure university environments and personal workstations are used.

3.2 System Design

This section details the system design of our vulnerability report generation pipeline, a visualisation can be found in Figure [fig. 1]

Dataset Curation and Preprocessing To ensure that the system is able to generalise better to real-world scenarios and go beyond basic pattern matching, we made the decision to adopt a dual-dataset strategy. Our primary dataset, TitanVul [14], was selected to act as both the training data for fine-tuning, and testing data. This is due to its sample size, constructed from an aggregation of seven vulnerability datasets, diversity across languages, thorough deduplication process, and providing a critical feature set required for our annotation pipeline. Our justification for choosing a selectively tougher but realistic dataset was so that the model could be taught to learn complex patterns and reason semantically

when analysing vulnerable code. Our second dataset, LLMVul, was acquired from HuggingFace. Deeper analysis of LLMVul revealed the dataset's synthetic nature. It lacked the noise and complexity of real-world code, hence providing similar patterns across vulnerabilities. Due to these findings, we explicitly kept this dataset for evaluation purposes only, which allows us to properly measure whether a model is able to generalise to real-world scenarios using TitanVul, or rely on simple pattern-matching. To define a research scope and ensure balanced class representation, we constrained the training and testing dataset to the Top 25 CWE types of 2025. Following annotation, we employed a stratified dataset splitting strategy on `cwe_id` labels, which, unlike random splitting, ensures that vulnerability classes are well represented across both sets. This also prevents label prediction bias and skewed performance results during inference and evaluation.

Model Selection This project makes use of two open-weight code specialised LLMs from the Qwen family [11]. Our system adopted a Teacher-Student framework [19], where we selected `Qwen2.5-Coder-32B-Instruct-GPTQ-Int4` for synthetic data annotation due to a higher-parameter count and code reasoning capabilities. This establishes a higher baseline for the student model to learn from. We specifically used the quantized version rather than the full-sized model because during preliminary experimentation, Out-Of-Memory (OOM) errors were encountered when running on our hardware. Our selected student model, Qwen2.5-Coder-7B-Instruct was preferred due to its specialisation in code, owing to it's pre-training, making it more suitable to understand and identify vulnerability patterns in code and reason about them. The model was also selected as it was an open-weight model that is smaller in size, allowing for quicker inference times and a feasible PEFT process on limited resources.

Synthetic Data Annotation Pipeline One of the main requirements of our system is for evaluated data objects to match a 6-field JSON vulnerability report containing the fields of `cwe_type`, `explanation`, `exploit_path`, `impact`, `mitigation`, and `sink_or_region`. Our curated datasets contain vulnerable code, patched code and cwe_type, however, they lack analysis and explanations. Without these key fields, the PALADIN framework cannot be applied in the evaluation phase. To address this shortcoming, we designed a Teacher-Student annotation pipeline to synthesise the missing reasoning fields. The Teacher model is tasked with analysing the difference in the function pairs along with other data to produce a vulnerability report drawn from it's advanced reasoning capabilities. While manual annotation was an option, due to the large amount of data samples, it was unfeasible for this project scope. Additionally, manual methods are prone to human mistakes and possible inconsistencies in the quality of annotations while automated annotation is able to adhere to the JSON schema.

Prompt Engineering and Schema Definition We employed a common prompt template across annotation and inference pipelines, requiring a JSON format output as mentioned before. Our prompt is designed to be reusable encompassing the role of a cybersecurity expert, providing task instructions, and output instructions, as well as CWE mapping context where applicable, shown in Listing 1. The purpose of role definition within prompt design is to encourage security-oriented reasoning and set constraints on tone, precision and factuality, while output instructions served as a guidance to the LLM on how to fill in the output template. User prompts were then fed into the LLM to begin the generation process, as the process is sequential and is dependent on the initial system prompt. For annotation on the TitanVul dataset, as cwe_id was provided by the creators, the CWE description was passed in dynamically into the user prompt shown in Listing 2) and cwe_id injected into the output rather than relying on the LLM's guess, saving tokens and lowering memory consumption. In LLMVul's scenario, as the dataset was not constricted to the Top 25 CWEs and did not have extra metadata like TitanVul, we excluded the CWE mapping context from the system prompt and user prompt shown in Listing 3 only contained in the vulnerable function, language and hint (short explanation of the vulnerability present in the code). During inference, to simulate both task-specific and realistic scenarios, we appended a short sentence onto the end of the task instructions highlighting the LLM having the ability to choose between safe or unsafe when inferencing.

QLoRA Fine-Tuning To adapt our base model onto the task under our hardware constraints, we adopted the PEFT strategy. Using LoRA, which freezes the pre-trained model weights and injects trainable rank decomposition matrices into the layers of the Transformer architecture [10]. This process reduces the number of trainable parameters significantly while still maintaining performance improvements similar to full-parameter fine tuning and is a widely used method that requires less GPU memory and time. While standard LoRA was the original plan, initial experiments revealed that loading the 16-bit base model onto our available hardware causes Out-Of-Memory (OOM) errors as we exceeded memory capacity. Therefore, we pivoted to QLoRA, which shrinks the size of the base model using 4-bit quantization, significantly reducing the memory needed. Despite this shrinking process, studies have reported that QLoRA is able to achieve performance improvements on par with standard methods, although with a trade-off in increased training time [5].

Inference Strategies To evaluate the model's ability for generating the vulnerability reports, the inference pipeline is designed to support the two inference strategies of zero-shot and few-shot prompting. Zero-shot was utilised to assess the model's base ability, allowing it to rely solely on pre-training knowledge and selected prompt engineering as described earlier. Inference results from zero-shot prompting served as the basis for comparison in our experiments involving different strategies and prompt designs. Another crucial inference strategy is few-shot prompting, which activates in-context learning. This method injects examples of the expected input and output into the LLM's context window, essentially mimicking conversations where the LLM captures the ground truth accurately. This guides the LLM's reasoning process by referencing previously seen examples without updating model weights or incurring a heavy computational cost.

3.3 System Implementation

This section details the technical implementation of our research project's system, which utilises the `PyTorch` framework and HuggingFace platform. Computationally intensive tasks were executed

on the University of Southampton's Iridis X HPC service, on the `ecsstudents_l4` partition compute nodes, which were equipped with 2 x NVIDIA L4 GPUs (with 24GB VRAM each). Access was managed via OpenOnDemand for Jupyter Notebooks, and SSH tunneling for batch job submissions through our personal laptops/desktops.

Dataset Curation and Preprocessing The source dataset, TitanVul initially contained 38,863 vulnerability-fix function pairs [14]. Data preprocessing involved feature extraction and filtering, where we selected the five most relevant features, namely `func_before`, `func_after`, `cwe_id`, `cve_description` and `extension`. To align with the design scope, the dataset was filtered to include only rows that are within the Top 25 CWEs of 2025, resulting in a dataset with 9212 samples.

Dataset Splitting The resulting annotated dataset was then divided into train and test sets following an 80/20 split. We then applied a stratified sampling strategy based on the `cwe_type` label, ensuring all vulnerability classes were well represented and equally distributed across both splits.

Synthetic Data Annnotation As our system design requires a 7-field JSON object for evaluation, we implemented a synthetic data annotation pipeline [19] using a teacher model (`Qwen/Qwen2.5-32B-Instruct-GPTQ-Int4`). Inference was accelerated using `vllm` engine, and to enforce the output format, we used a `pydantic` BaseModel inserted into the `structured_outputs` argument of the sampling parameters using a `StructuredOutputsParams` object. Considering the length of the source code functions and the fully built prompt as a whole, the context window was set to 12,288 tokens and a maximum output length of 2,048 tokens. As the basis of our work relies on factuality and precision, the LLM was configured to use a temperature of 0.1 and a set seed of 42 to ensure reproducibility. Due to LLM's probabilistic generation nature, we encountered a small failure rate on annotation of approximately 0.12% (11 failures out of 9,212 samples) in the TitanVul dataset.

QLoRA Fine-Tuning Configuration The model was fine-tuned using the QLoRA technique to optimize memory efficiency. We relied on the HuggingFace `transformers`, `peft` and `trl` libraries to configure the trainer. The base model `Qwen-2.5-Coder-7B-Instruct`, was loaded in 4-bit precision using `nf4` quantization and `bf16` for computations. The LoRA configuration was set as follows:

- **Rank ($r = 16$):** Selected to ensure the model has sufficient capacity to capture complex structures and patterns present in vulnerable code. This setting aligns with prior literature in this domain from Mao et al .[19].
- **Alpha ($\alpha = 32$):** Adhering to the standard heuristic of $\alpha = 2 \times r$ [4]
- **Dropout ($dropout = 0.05$):** applied a dropout rate of 0.05 to the LoRA layers to prevent overfitting during fine-tuning
- **Target Modules:** All linear layers (`q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, `down_proj`) were targeted, as it has been demonstrated to match the performance of full fine-tuning [5].

Training Hyperparameters Due to the VRAM constraints of the L4 GPUs, training hyperparameters were selected through a trial and error process. Some key hyperparameters include:

- **Batch Configuration:** A per-device batch size of 1 was used with gradient accumulation of 8 steps, yielding in an effective batch size of 8
- **Optimizer:** `paged_adamw_32bit`, that uses paging techniques to offload optimizer states to CPU RAM, reducing GPU memory consumption
- **Scheduler:** Training proceeded for 3 epochs using a cosine learning rate scheduler with a warmup ratio of 3%.
- **Learning Rate & Training Stability:** The learning rate was set to 2×10^{-5} with a maximum gradient norm of 0.3. In earlier experiments with standard learning rates (2×10^{-4} and 1×10^{-4}), training was divergent and had gradient explosion, resulting in model collapse, even after gradient clipping was applied. Reducing the rate to 2×10^{-5} was required to ensure stable convergence.

The fine-tuning process was completed in $\sim$ 9 hours and 43 minutes on the dual NVIDIA L4 GPU configuration. A graph detailing the fine-tuning progress is included in the appendix [fig. 2].

Inference The inference pipeline is very similar in terms of design and implementation with the annotation pipeline with minor adjustments to the configuration. We replaced the annotation prompts with various inference prompts while retaining the original temperature setting to prioritise factuality over creativity. To ensure efficient memory utilisation, the context window was reduced to 8,192 tokens due to the shorter prompt lengths for inference. Output tokens remained at 2,048 to ensure consistency in ground truth and LLM predicted vulnerability reports. Finally, we applied the same JSON schema removal protocol used in the annotation phase.

4 Evaluation/Results

4.1 Evaluation methods

In this paper, we propose the PALADIN framework (**P**rogrammatic **A**nalysis & **L**LM **A**ssessment for **D**efence **In**telligence), a novel evaluation protocol designed to rigorously benchmark Large Language Models on their end-to-end security reasoning capabilities.

Unlike existing benchmarks that primarily focus on binary vulnerability classification, PALADIN utilises the model's decomposed output to compare its predictions with the ground truth accurately. These fields cover all critical dimensions of security expertise: taxonomic precision (CWE accuracy), semantic depth (explanation and impact similarity), remedial viability (mitigation effectiveness), and exact extraction of vulnerable code snippets. The PALADIN framework utilises high-resolution similarity scoring using a reliable sentence transformer (all-MiniLM-L6-v2). Combining Jaccard similarity enables us to reliably verify whether the extracted code snippet matches the ground truth.

By evaluating and critiquing an LLM's output on each of the following fields, we can push our LLM into completing more tasks and solving more problems in the cybersecurity field:

(1) **CWE Exact Accuracy** - How accurate an LLM is at identifying the correct vulnerability in a function.

(2) **Explanation Mean Similarity & Explanation Pass Rate** - How correct the explanation is, shows how useful it is for a person to read the explanation about a spotted vulnerability.
(3) **Exploit Path Mean Similarity & Exploit Path Pass Rate** - How accurate the crafted exploit is, can lead to creating proof of concepts.
(4) **Impact Mean Similarity & Impact Pass Rate** - How well the model understands the main repercussions from the vulnerable code.
(5) **Mitigation Mean Similarity & Mitigation Pass Rate** - How accurate the model is at patching vulnerable code, can lead to automated vulnerability patching pipelines in projects.
(6) **Sink or Region Mean Similarity & Sink or Region Pass Rate** - Whether the model can find the exact code snippet that causes the vulnerability, helps with patching vulnerabilities.
(7) **Mean Final Score & Complete Report Pass Rate** - Mean Final Score gives a rating out of 0-1.0 of how the model performed in all of the fields; it uses a sum of weighted components, determined by how important each field is to the overall accuracy of the model output. Complete report pass rate is the percentage of predictions in which all fields passed the correctness threshold. These two give an overall rating to the usefulness of an LLM.

An example evaluation report for an LLM's predictions can be found in Appendix B.1

4.2 Taxonomic Classification Accuracy

Objectively, the most critical metric for any static vulnerability detection system is the frequency with which it correctly identifies the vulnerability in the function it analyses. This section discusses the accuracy of each method in determining the exact vulnerabilities present in the code.

The table in Appendix B.3 shows the results of the exact CWE accuracy of the model's predictions

4.2.1 Dataset difficulty The difference in dataset difficulty is clearly shown. All methods consistently score over 30% lower on TitanVul than LLMVul. TitanVul contains a diverse set of vulnerabilities, many of which led to the creation of new Common Vulnerabilities and Exporsures (CVE) entries; testing on this dataset effectively evaluates the LLM's effectiveness at finding zero-days in the code.

This significant difference in scores suggests that the LLM's ability to handle complex and rare vulnerability patterns is weaker than its ability to handle straightforward, common vulnerability patterns, indicating that the LLM is less effective at understanding the actual semantics of code.

4.2.2 Traditional SAST Semgrep scores very low across both datasets, indicating that traditional SAST methods especially struggle with complex and subtle vulnerabilities. SAST focuses on predefined pattern rules, which are ineffective in such a sparse and complex dataset as TitanVul. It performs significantly better on LLMVul, which contains many repeated, simple vulnerability patterns. It also shows that traditional SAST tools aren't nearly as effective and wouldn't be sufficient for reliable vulnerability detection.

4.2.3 Realism of "With None" constraint We observe that the rejection logic holds firmly in LLMVul, with accuracy barely dropping; however, in TitanVul, it drops significantly. LLMVul contains simple, standard patterns that the model appears confident in, but in the diverse environment of TitanVul, the model often defaults quickly back to "None" (False Negatives).

Looking at graphs [fig. 14] and [fig. 16] in Appendix D, we can see the model guesses CWE-89 significantly more, suggesting that with its new none option, it chooses only vulnerability types it feels confident in; otherwise, it defaults back to none. CWE-89 is a very 'famous' vulnerability and the model would have initially been trained on it a lot, hence why it would feel more confident answering CWE-89.

In more realistic scenarios, this suggests that the LLM's confidence primarily determines its results. Simple classifications don't reduce accuracy; however, more complex vulnerabilities cause the LLM to select "None" rather than remain unsure of the answer. This is suboptimal for vulnerability detection, as most systems prefer false positives to false negatives, whereas the cautious LLM does the opposite.

4.2.4 Hope in fine-tuning Looking at TitanVul, we observe a significant performance improvement with the fine-tuned model. This is a promising sign that LLMs can capture the subtle features and complex semantics of real-world code examples. Furthermore, adding the "With None" constraint appears to yield only a slight drop, suggesting that fine-tuning also helps the model become more confident in identifying vulnerabilities.

The fine-tuned model underperforms on LLMVul, which could indicate overtraining; however, LLMVul contains many vulnerabilities that extend far beyond the top 25 CWEs. The model attempts to force these **OOD (out-of-distribution)** vulnerabilities into the categories it was trained on. This was shown in [15] that LLM-based vulnerability detection has a significant lack of generalisation capability for OOD vulnerabilities.

Given the scope of the project, we limited training to the top 25 CWEs. This could serve as proof of concept that with a larger training dataset, the LLM would perform much better on these vulnerability types.

Fine-tuning yields substantial improvements in vulnerability detection and in understanding complex, subtle indicators, as evidenced by increases in accurate classifications on difficult test cases.

4.2.5 Zero-shot vs Few-shot As a reminder: TitanVul few-shot examples were 79, 787, and 20. LLMVul few-shot examples were 120, 787, 89.

In most cases, few-shot prompting seems to increase accuracy very slightly, except for few-shot with none option, which significantly decreases the accuracy (~14% to 5%). However, comparing the frequency of each CWE type in the model's predictions between zero-shot and few-shot settings, we observe a notable difference. Looking at [fig. 6] vs [fig. 8] in Appendix D, CWE-787 was predicted almost 10 times more in zero-shot than in few-shot. CWE-787 (Out-of-bounds write) was among the examples provided to the model, suggesting that the model exhibits **majority label bias**. This is when the distribution of labelled examples in the in-context samples skews the distribution of predictions [8]. Typically, larger

models (70b) show increased robustness to majority label bias compared to smaller models (7b) like the one we used, so it's likely this is what's happening; this is due to the fact that smaller models have an over-reliance on the distribution of in-context examples (low-level patterns) rather than the subtle semantics [8].

Looking at [fig. 14] vs [fig. 16] in Appendix D, we see that the opposite has happened, and CWE-89 has spiked drastically even though it wasn't part of the few-shot examples. Some sources suggest that when introducing ambiguity or complex constraints such as a "None" option, the model defaults to its pre-training rather than the task at hand. It was also shown that if a specific class label was highly prevalent in the dataset the model was originally trained on, the model will have a strong "**uncontextual preference**" to output that token [28]. CWE-89 (SQL Injection) is among the most well-documented vulnerabilities and is likely to appear in most LLM training sets significantly more often than most other vulnerabilities.

Lastly, observing differences between zero- and few-shot on LLMVul (e.g. [fig. 18] vs [fig. 20]), we note that all CWE types in the examples consistently decrease in model prediction frequency. It's unclear what exactly the reason could be, however it might suggest that the model becomes a lot more confident in classifying that specific CWE as the accuracy for the given CWE types don't change but the frequency of predicting it decreases significantly.

4.3 Vulnerability Localization and Sink Identification

A key metric for assessing the model's reliability is its accuracy in identifying the exact snippet of code that causes the vulnerability. This is a critical task for the LLM to get right to reliably patch vulnerable code.

The table below shows the pass rate for finding the vulnerable region, given that the CWE was predicted correctly.

4.3.1 Increased precision from the "None" option An interesting observation is that adding the "None" option serves as a precision filter. By allowing the model to reject ambiguous cases, the subset of samples that the model commits to is of much higher quality. This shows that providing a 'back-out' option can increase the model's confidence when assigning vulnerabilities, yielding much higher precision. Without a "None" option, LLMs tend to exhibit a "**bias towards action**", which means they will confidently select the most plausible distractor rather than admit they do not know the answer [25].

4.3.2 Finetuning's specialisation penalty Fine-tuning has been shown to consistently decrease the LLM's ability to find the vulnerable code snippet. It's clear that fine-tuning has increased its ability to label the data correctly, however, it seems to have decreased its spatial reasoning.

This seems to indicate that the fine-tuned model undergoes **format specialisation** and **over-specialisation** when it becomes overly focused on the specific data composition and output structure during fine-tuning [26]. The `sink_or_region` field in TitanVul's dataset will have highly disparate values. If the model overfits to these, then all data it is tested on will be OOD, causing it to underperform. This is a well-documented and common phenomenon observed with fine-tuned models, suggesting that LLMs cannot understand the subtle underlying mechanisms of vulnerability detection and can only identify general, unimportant patterns.

4.4 Qualitative Interpretability

This section focuses on the model's explanations, exploit path, mitigation, and its ability to understand the vulnerability and its impact on the code. All data and tables for this section are available in Appendix B.2.

4.4.1 High Accuracy in Determining Exploit Path All models consistently achieve higher exploit-path accuracy than in every other category. This is likely because exploit paths follow a predictable pattern that the LLM can use as a template in its predictions.

Explanation requires a higher level of abstract reasoning that connects the code's intent to the security risk, which is a hard task for the model. This implies that, once again, the model fails to grasp this deeper understanding and instead relies on non-essential, predictable patterns.

4.4.2 Low Impact Score This idea is reinforced by the significantly lower impact score. Determining the severity and consequences of the security risk is the hardest part of the security workflow for the model. The model struggles to understand why this matters in the broader context.

4.4.3 The Difficulty of Mitigation for Finetuned Models The fine-tuned model consistently scores lower on mitigation, especially compared to "Few-Shot with None." Mitigation fields in the dataset are often code-specific and require a deep understanding of the exploit and vulnerability area.

Specialised fine-tuning is a discriminative task (e.g. classifying an example); however, mitigation is a creative task. By fine-tuning and possibly overfitting, we lose the flexibility needed to 'imagine' a fix. This is a trade-off that comes with **specialised fine-tuning** vs. **instruction tuning**

4.4.4 Explainability It has been shown that the model improves the quality of explanations after fine-tuning. It's likely that this is because the model learns a lot of highly cybersecurity-specific terminology, which then scores higher on the evaluator as it begins to use keywords from the training dataset.

It's also notable that mitigation scores decreased after fine-tuning, as we are reducing the model's creative capabilities, whereas explanation scores increased. This is likely because explaining a vulnerability is often a pattern-recognition task; mapping certain CWEs to specific keywords would likely perform well against the ground truth.

4.5 RQ1: How effectively can LLMs detect web and cloud vulnerabilities using prompt-based and peft methods?

4.5.1 Exact CWE Identification The effectiveness of detection varies significantly with code complexity. In simple environments such as LLMVul, the model performs exceptionally well; however, in more complex environments that require a deep understanding of the

code, such as TitanVul, identifying the exact CWE remains a major challenge.

- **Prompt-Based:** Few-Shot is shown to be an unreliable method when attempting to increase the model's performance. It leads to over-reliance on in-context data and substantially skews the distribution of CWE types in the model's predictions. It unpredictably changes the accuracy, indicating that it heavily interferes with the reasoning the model learnt during pretraining.
- **PEFT (Fine-tuning):** Fine-tuning manages to provide a consistent increase in performance for labelling, scoring ~10% more than zero- and few-shot prompting in TitanVul. However, this trend didn't continue with LLMVul, suggesting that it struggles with out-of-distribution data, likely due to overfitting.

4.5.2 Vulnerability Localisation Once a model correctly identifies a CWE, its ability to pinpoint the vulnerable code is an important metric for vulnerability detection. The model performs extremely well on LLMVul, which has simple vulnerability patterns, whereas it drops off substantially on TitanVul, which is much more difficult. Fine-tuning doesn't seem to help, as it underperforms; this is likely because the model loses important "spatial reasoning" needed to precisely locate the security flaw.

4.5.3 In Conclusion LLMs are highly effective at detecting and labelling simple vulnerabilities (LLMVul); however, they struggle with complex vulnerabilities that require a deeper understanding (TitanVul). Fine-tuning yields consistent, promising results for labelling complex vulnerabilities and suggests that, with more time and data, it could accurately capture subtle patterns; however, this comes at the cost of "localisation intelligence", optimising the model to become a better labeller but a worse "spotter" of the exact vulnerable code.

4.6 RQ2: How well can LLMs generate accurate, complete, and actionable explanations on detected vulnerabilities?

LLM adaptation strategies (e.g., prompting and specialised fine-tuning) have been shown to increase accuracy, and specific combinations yield promising results. Often, results point to an improvement in semantic pattern matching rather than to an increase in the models' ability to "grasp" the underlying vulnerability.

4.6.1 The "Semantic Ceiling" in Prompt-Based Methods Zero-shot and few-shot methods reveal that LLMs possess an inherent "security common sense" from pre-training; however, they struggle with complex reasoning. Beyond vulnerability classification in code, many metrics show only marginal improvements, if any, when fine-tuned or using different prompting techniques, and in some cases even decrease accuracy. Indicating the model is already at its best from pre-training.

- **Zero-Shot Stability:** Zero-shot's performance is remarkably stable across every metric, even barely decreasing from adding the "None" option. This suggests the model relies on simple pre-trained patterns of common flaws like SQL injection (SQLi) or `eval`
- **Few-Shot Anchoring:** Adding examples often degrades performance, either causing the model to over-choose the example or completely ignoring it when making predictions. This suggests that the drastic frequency changes are due to the model's over-reliance on in-context patterns, indicating that it cannot use the reasoning behind the examples.

4.6.2 Misdirection from Fine-tuning Fine-tuning provides the highest exact CWE accuracy, a promising result for taxonomic classification; however, it often compromises the model's broader reasoning and problem-solving. This was shown in the performance drop-off in determining the mitigation strategy. This indicates that fine-tuning makes the model a better classifier but a worse reasoner; it has "stiffened" and lost vital reasoning ability.

4.6.3 The Gap Between Localisation and Reasoning Although the model scores highly and identifies the correct vulnerable code, it struggles to analyse it properly (very low impact score). This is an important indicator that they lack a deep understanding, as vulnerable code often exhibits specific patterns that even traditional SAST scanners can detect. The model can detect 'dangerous-looking' code, yet cannot confidently determine the semantic consequences of that vulnerability.

4.6.4 In Conclusion It's clear that LLMs are effective at pattern-based detection and that fine-tuning teaches the model key features to look for; however, they cannot grasp the deep semantic logic required to analyse complex vulnerabilities that go beyond simple concepts. This inability to improve understanding raises concerns about the reliability of LLMs to deliver actionable results.

5 Discussion

Discussion Due to the novelty of our proposed evaluation framework, direct metric and quantitative comparisons with existing studies are limited. However, our findings can be put into the context of literature to examine trends amongst studies using similar techniques. We observe the same performance decrease consistent with the findings of Ding et al. [6], who reported a drop from 68.26% F1 score on the older BigVul dataset to a 3.09% F1 score on the deduplicated PRIMEVUL. Their reason for this degredation in performance is largely due to the existence of duplicate functions and data leakage into test sets on BigVul leading to artificially inflated performance scores, which are present in older studies. Our experiments reflect similar performance sensitivity to dataset as the paper, our model scored a significantly higher CWE accracy on the synthetic LLMVul dataset compared to TitanVul across all configurations. This reinforces the necessity of high-quality real-world simulated data to prevent false confidence in model ability in security centric tasks.

Limitations and Future Work We have shown evidence that prompting strategies and QLoRA fine-tuning can improve a model's ability to detect and generate reasoned vulnerability reports answering our first research question. However, our findings are currently applicable to one LLM size (7B). Due to the computational constraints, we leave exploration of varying model sizes (3B, 70B, among another variants) to future work. Another question is on

the topic of model specialisation. As our work utilises a coder specialised model from the Qwen family, a possible avenue of future research will be to expand into a comparative study across models with different pre-training backgrounds. Specifically, comparing general-purpose LLMs against reasoning-focused models or vulnerability specialists LLMs.

Another limitation acknowledged is the usage of a 32B 4-bit quantized LLM for the automated annotation process. While the model has been established to be superior over our base model, we observe that annotation quality can be inconsistent across data samples, occasionally mislabeling CWE types, overall leading to a less reliable dataset. Future work should involve usage of a stronger model with better reasoning capabilities accompanied with human-review to verify the quality of the resulting annotations to facilitate a higher-quality dataset.

The primary limitation of the PALADIN framework is that semantic similarity does not imply functional correctness. Ideally, we would use sentence embeddings optimised for cybersecurity applications. Certain keywords can completely change the meaning of sentences, rendering the security analysis functionally useless, yet they would still score highly. The evaluator is also unable to group similar CWEs. If a bug is identified as CWE-79 (XSS) but the model predicts CWE-89 (Basic XSS), it would be flagged as a failure; with a larger knowledge base, this could be avoided, but this was infeasible given the scope of this project.

6 Improvements/Changes

The constructive feedback provided by the module team resulted in a few changes to the structure of the report when compared with the mid-term submission. To address the feedback, we first clarify the novelty of our project within the introduction section. In the mid-term report, while we attempted to set the scene by outlining existing tools and approaches, discussing their respective strengths/weaknesses, and reviewing recent research in our field, we failed to provide a concrete definition of our proposed system. As a result, the novelty of our contribution was not made sufficiently explicit. In the current report, this issue is addressed by providing a high-level, but explicit definition of our proposed **PALADIN evaluation framework** in the introduction, followed by a more thorough and formal specification in Section 4.

Specifically within the related work, as the LLM-based approaches section was identified as more interesting and relevant than the pre-LLM sections, we restructured section 2.3 into two distinct sections, improving focus and clarity. To first address concerns regarding the reference quality, the final report prioritises the use of peer-reviewed academic literature wherever possible and removed low-quality references, such as Stack Overflow surveys. Second, we revised the focus of the content by minimising low-level technical descriptions of how different vulnerability detection approaches operate internally. Instead, we explored relevant literature that examines how these approaches perform in practice, where they fall short in their ability to reason about and explain vulnerabilities, paving the way for our report.

Regarding changes in the project scope from the original proposal, we found that the scope of the problem was not clearly defined instead focusing on a methodology and implementation plan. Consequently, we implemented a number of changes, first of which was a replacement in the primary dataset which reasons are justified within the 3.2. The second change was regarding the method used to implement PEFT. we initially planned to use LoRA fine-tuning, however switched to QLoRA. The primary reason for this change was due to the lack of VRAM available on our hardware.

7 Individual Contribution

Brandon (27%): Designed the vulnerability report generation system encapsulating the three pipelines of data preprocessing, annotation and inference inclusive of model selection, prompt design and dataset selection. Designed and coded up the LoRA fine-tuning script, with careful selection of LoRA and training hyperparameters. Documenting the entire process and justification within Section 3.2 and 3.3 of the report.

Dan (27%): Designed the evaluation framework algorithm and LLM schema, aided in dataset annotation. Ran and collated results for LLM prompting methods, finetuning and SAST on test datasets. Researched, evaluated, and reasoned the collected results. Wrote section 4.

Eren (23%): Described/written the threat model for the system, revised parts of the related work section and added to it, written the improvements/changes and conclusion sections, general report structure fixes.

Jonathan (23%): Written parts of introduction, section 2.1, system design model selection and prompt engineering, QLoRa fine tuning, and general report structure fixes.

8 Conclusions

This comparative study mainly investigated the effectiveness and limitations of LLMs for web and cloud vulnerability detection, with a particular focus on their reasoning and explainability capabilities, while also sizing their performance relative to traditional SAST tools. We introduced PALADIN, a novel evaluation framework that moves beyond binary vulnerability classification by assessing multiple dimensions of security expertise, including taxonomic accuracy, vulnerability localisation, explanation quality, impact understanding, and mitigation viability. Through several phases of evaluation across realistic and synthetic datasets, we demonstrated that, even though LLMs outperform traditional SAST tools on simpler, pattern-driven vulnerabilities, their struggles with semantically rich, real-world cases continue. Prompt-based methods showed limited and varying gains, whereas PEFT improved the accuracy of classification at the expense of localisation and reasoning flexibility. These results hint that current LLMs fail to get over the threshold to become robust security reasoners, rather staying as more focused pattern recognisers. By revealing systematic gaps between surface-level correctness and true security understanding, PALADIN hints at the need for future work that stretches across model variants, strengthens annotation reliability, and develops evaluation metrics that move beyond semantic similarity, with functionally correct, CWE-aware security reasoning in sight.

References

[1] Bader Alouffi, Muhammad Hasnain, Abdullah Alharbi, Wael Alosaimi, Hashem Alyami, and Muhammad Ayaz. 2021. A Systematic Literature Review on Cloud Computing Security: Threats and Mitigation Strategies. *IEEE Access* 9 (2021), 57792–57807. https://doi.org/10.1109/ACCESS.2021.3073203

[2] Jung Hyun An, Zhan Wang, and Inwhee Joe. 2023. A CNN-based automatic vulnerability detection. *EURASIP Journal on Wireless Communications and Networking* 2023, 1 (May 2023), 41. https://doi.org/10.1186/s13638-023-02255-2

[3] Murat Aydos, Çiğdem Aldan, Evren Coşkun, and Alperen Soydan. 2022. Security testing of web applications: A systematic mapping of the literature. *Journal of King Saud University - Computer and Information Sciences* 34, 9 (Oct. 2022), 6775–6792. https://doi.org/10.1016/j.jksuci.2021.09.018

[4] Dan Biderman, Jacob Portes, Jose Javier Gonzalez Ortiz, Mansheej Paul, Philip Greengard, Connor Jennings, Daniel King, Sam Havens, Vitaliy Chiley, Jonathan Frankle, Cody Blakeney, and John P. Cunningham. 2024. LoRA Learns Less and Forgets Less. https://doi.org/10.48550/arXiv.2405.09673 arXiv:2405.09673 [cs].

[5] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. https://doi.org/10.48550/arXiv.2305.14314 arXiv:2305.14314 [cs].

[6] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability Detection with Code Language Models: How Far Are We? https://doi.org/10.48550/arXiv.2403.18624 arXiv:2403.18624 [cs].

[7] Katerina Goseva-Popstojanova and Andrei Perhinschi. 2015. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology* 68 (Dec. 2015), 18–33. https://doi.org/10.1016/j.infsof.2015.08.002

[8] Karan Gupta, Sumegh Roychowdhury, Siva Rajesh Kasa, Santhosh Kumar Kasa, Anish Bhanushali, Nikhil Pattisapu, and Prasanna Srinivasa Murthy. 2023. How Robust are LLMs to In-Context Majority Label Bias? arXiv:2312.16549 [cs.LG] https://arxiv.org/abs/2312.16549

[9] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. 2024. Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey. *Transactions on Machine Learning Research* 2024 (2024). https://www.scopus.com/inward/record.uri?eid=2-s2.0-105000099444&partnerID=40&md5=61eb7331da670a50300863fa23325197 Cited by: 19.

[10] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. https://doi.org/10.48550/arXiv.2106.09685 arXiv:2106.09685 [cs].

[11] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (2024).

[12] Omar Javed and Salman Toor. 2021. An Evaluation of Container Security Vulnerability Detection Tools. In *2021 5th International Conference on Cloud and Big Data Computing (ICCBDC)*. ACM, Liverpool United Kingdom, 95–101. https://doi.org/10.1145/3481646.3481661

[13] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2023. Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities. *arXiv preprint arXiv:2311.16169* (2023). https://arxiv.org/abs/2311.16169 Accessed 2025-10-30.

[14] Yikun Li, Ngoc Tan Bui, Ting Zhang, Martin Weyssow, Chengran Yang, Xin Zhou, Jinfeng Jiang, Junkai Chen, Huihui Huang, Huu Hung Nguyen, Chiok Yew Ho, Jie Tan, Ruiyin Li, Yide Yin, Han Wei Ang, Frank Liauw, Eng Lieh Ouh, Lwin Khin Shar, and David Lo. 2025. Out of Distribution, Out of Luck: How Well Can LLMs Trained on Vulnerability Datasets Detect Top 25 CWE Weaknesses? https://doi.org/10.48550/arXiv.2507.21817 arXiv:2507.21817 [cs].

[15] Yue Li, Xiao Li, Hao Wu, Minghui Xu, Yue Zhang, Xiuzhen Cheng, Fengyuan Xu, and Sheng Zhong. 2025. Everything You Wanted to Know About LLM-based Vulnerability Detection But Were Afraid to Ask. arXiv:2504.13474 [cs.CR] https://arxiv.org/abs/2504.13474

[16] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings 2018 Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2018.23158 arXiv:1801.01681 [cs].

[17] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*. 74–81. https://aclanthology.org/W04-1013/

[18] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. 2020. Software Vulnerability Detection Using Deep Neural Networks: A Survey. *Proc. IEEE* 108, 10 (Oct. 2020), 1825–1848. https://doi.org/10.1109/JPROC.2020.2993293

[19] Qiheng Mao, Zhenhao Li, Xing Hu, Kui Liu, Xin Xia, and Jianling Sun. 2024. Towards Explainable Vulnerability Detection with Large Language Models. *arXiv preprint arXiv:2406.09701v5* (2024). https://arxiv.org/abs/2406.09701v5

[20] MITRE Corporation. 2024. Common Weakness Enumeration (CWE): The Standard for Software Weaknesses. https://cwe.mitre.org. Accessed 2025-10-30.

[21] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proc. ACL*. 311–318. https://doi.org/10.3115/1073083.1073135

[22] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Oz demir, Paul Ellingwood, and Marc McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 757–762. https://doi.org/10.1109/ICMLA.2018.00120

[23] Alexey Shestov, Rodion Levichev, Ravil Mussabayev, Evgeny Maslov, Anton Cheshkov, and Pavel Zadorozhny. 2024. Finetuning Large Language Models for Vulnerability Detection. https://doi.org/10.48550/arXiv.2401.17010 arXiv:2401.17010 [cs].

[24] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. 2025. LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning. https://doi.org/10.48550/arXiv.2401.16185 arXiv:2401.16185 [cs].

[25] Zhi Rui Tam, Cheng-Kuang Wu, Chieh-Yen Lin, and Yun-Nung Chen. 2025. None of the Above, Less of the Right: Parallel Patterns between Humans and LLMs on Multi-Choice Questions Answering. arXiv:2503.01550 [cs.CY] https://arxiv.org/abs/2503.01550

[26] Yihan Wang, Si Si, Daliang Li, Michal Lukasik, Felix Yu, Cho-Jui Hsieh, Inderjit S Dhillon, and Sanjiv Kumar. 2024. Two-stage LLM Fine-tuning with Less Specialization and More Generalization. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=pCEgna6Qco

[27] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. BERTScore: Evaluating Text Generation with BERT. In *ICLR*. https://arxiv.org/abs/1904.09675

[28] Tony Z. Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-Shot Performance of Language Models. arXiv:2102.09690 [cs.CL] https://arxiv.org/abs/2102.09690

[29] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2025. Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025). https://doi.org/10.1145/3708522 Publisher: Association for Computing Machinery.

[30] Xin Zhou, Ting Zhang, and David Lo. 2024. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, Lisbon Portugal, 47–51. https://doi.org/10.1145/3639476.3639762

A System Diagram

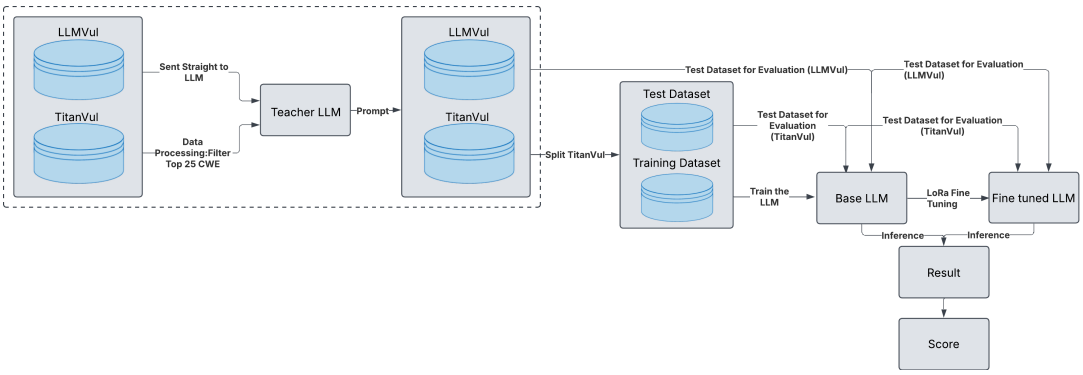

Figure 1: Diagram For System Pipeline

Listing 1: Reusable Prompt Template

```
You are a cybersecurity expert. Analyze the provided VULNERABLE FUNCTION and generate a structured vulnerability report.
Your goal is to be precise and accurate.
(Optional for None: If there is no vulnerability or the vulnerability doesn't match to these CWE KNOWLEDGE BASE then put None
     for all of the fields.)

CONTEXT: CWE KNOWLEDGE BASE
You are restricted to the following 25 CWEs. Use the provided descriptions to distinguish between them.
- "CWE-79": "Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')",
- "CWE-89": "Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')",
- "CWE-352": "Cross-Site Request Forgery (CSRF)",
- "CWE-862": "Missing Authorization",
- "CWE-787": "Out-of-bounds Write",
- "CWE-22": "Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')",
- "CWE-416": "Use After Free",
- "CWE-125": "Out-of-bounds Read",
- "CWE-78": "Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')",
- "CWE-94": "Improper Control of Generation of Code ('Code Injection')",
- "CWE-120": "Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')",
- "CWE-434": "Unrestricted Upload of File with Dangerous Type",
- "CWE-476": "NULL Pointer Dereference",
- "CWE-121": "Stack-based Buffer Overflow",
- "CWE-502": "Deserialization of Untrusted Data",
- "CWE-122": "Heap-based Buffer Overflow",
- "CWE-863": "Incorrect Authorization",
- "CWE-20": "Improper Input Validation",
- "CWE-284": "Improper Access Control",
- "CWE-200": "Exposure of Sensitive Information to an Unauthorized Actor",
- "CWE-306": "Missing Authentication for Critical Function",
- "CWE-918": "Server-Side Request Forgery (SSRF)",
- "CWE-77": "Improper Neutralization of Special Elements used in a Command ('Command Injection')",
- "CWE-639": "Authorization Bypass Through User-Controlled Key",
- "CWE-770": "Allocation of Resources Without Limits or Throttling"]

OUTPUT INSTRUCTIONS:
1. cwe_type: The standard CWE ID (e.g., 'CWE-787').
2. explanation: A single technical sentence explaining the flaw. Strictly under 30 words. Do not repeat the CWE name.
3. exploit_path: A sentence describing the attack flow in a theoretical scenario of how an attacker triggers this.
4. impact: The consequence. Max 5 words (e.g., 'Remote Code Execution').
5. mitigation: Specific code fixes (e.g., 'Use strncpy instead of strcpy').
6. sink_or_region: The EXACT snippet of code from the VULNERABLE FUNCTION that causes the vulnerability.
```

Listing 2: TitanVul User Prompt Code Snippet

```
f"VULNERABLE FUNCTION:\n{batch_row.get('func_before', '')}\n\n"
f"PATCHED FUNCTION:\n{batch_row.get('func_after', '')}\n\n"
f"CWE ID:\n{cwe_id}\n\n"
f"CWE DEFINITION:\n{cwe_desc}\n\n"
f"CVE DESCRIPTION:\n{batch_row.get('cve_description', '')}\n\n"
f"LANGUAGE:\n{lang}"
```

Listing 3: LLMVul User Prompt Code Snippet

```
f"VULNERABLE FUNCTION:\n{batch_row.get('func_before', '')}\n\n"
f"PATCHED FUNCTION:\n{batch_row.get('func_after', '')}\n\n"
f"CWE ID:\n{cwe_id}\n\n"
f"CWE DEFINITION:\n{cwe_desc}\n\n"
f"CVE DESCRIPTION:\n{batch_row.get('cve_description', '')}\n\n"
f"LANGUAGE:\n{lang}"
```

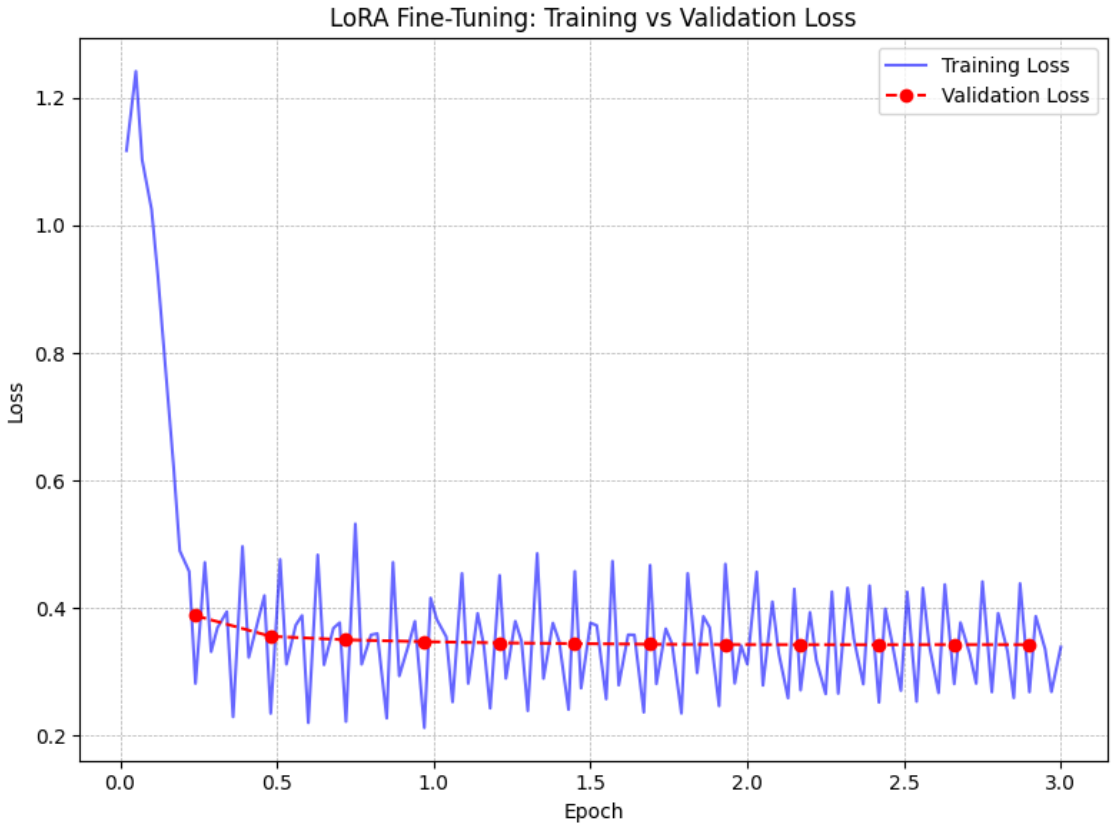

Figure 2: LoRA Fine-Tuning Progress Graph

B Evaluation Results

Table 1: Example Model Performance Metrics

Metric	Value
Complete Report Pass Rate	0.0256
CWE Exact Accuracy	0.1946
CWE Valid Rate	0.6087
Explanation Mean Similarity	0.6593
Explanation Mean Similarity (CWE Correct)	0.7663
Explanation Pass Rate @0.75	0.2076
Explanation Pass Rate @0.75 (CWE Correct)	0.6134
Exploit Path Mean Similarity	0.7774
Exploit Path Mean Similarity (CWE Correct)	0.8260
Exploit Path Pass Rate @0.75	0.5929
Exploit Path Pass Rate @0.75 (CWE Correct)	0.8067
Impact Mean Similarity	0.6302
Impact Pass Rate @0.7	0.2136
Mean Final Score	**0.2985**
Mean Final Score (CWE Correct)	0.8208
Mitigation Mean Similarity	0.6564
Mitigation Mean Similarity (CWE Correct)	0.7336
Mitigation Pass Rate @0.75	0.1847
Mitigation Pass Rate @0.75 (CWE Correct)	0.4538
Schema Completeness Rate	1.0000
Sink or Region Mean Similarity	0.6627
Sink or Region Mean Similarity (CWE Correct)	0.7348
Sink or Region Pass Rate @0.6	0.4905
Sink or Region Pass Rate @0.60 (CWE Correct)	0.6471

B.1 Evaluation Result comparisons

	TitanVul	LLMVul
Zero-Shot	61.345	69.161
Few-Shot	66.230	89.806
Zero-Shot with None	64.453	70.571
Few-Shot with None	82.796	90.533
Finetuned	68.478	88.126
Finetuned with None	65.177	87.04

Table 2: Comparison between the accuracies of how similar the explanation was to the ground truth, given they predicted the correct CWE (pass rate of similarity score@0.75).

	TitanVul	LLMVul
Zero-Shot	80.672	95.888
Few-Shot	89.005	96.721
Zero-Shot with None	87.5	96.358
Few-Shot with None	94.624	98.016
Finetuned	90.761	98.848
Finetuned with None	90.503	98.924

Table 3: Comparison between the accuracies of how similar the exploit path was to the ground truth, given they predicted the correct CWE (pass rate of similarity score@0.75).

	TitanVul	LLMVul
Zero-Shot	45.378	79.317
Few-Shot	39.791	82.892
Zero-Shot with None	45.312	80.339
Few-Shot with None	63.441	81.439
Finetuned	39.493	79.752
Finetuned with None	39.851	78.744

Table 4: Comparison between the accuracies of how similar the mitigation was to the ground truth, given they predicted the correct CWE (pass rate of similarity score@0.75).

	TitanVul	LLMVul
Zero-Shot	21.362	44.172
Few-Shot	22.725	52.24
Zero-Shot with None	15.016	44.71
Few-Shot with None	4.674	49.817
Finetuned	31.192	58.06
Finetuned with None	31.284	58.522

Table 5: Comparison between the accuracies of how similar the impact was to the ground truth (pass rate of similarity score@0.7).

B.2 Exact CWE Accuracy Results

	TitanVul	LLMVul
Semgrep (SAST)	3.226	33.135
Zero-Shot	19.455	52.402
Few-Shot	20.817	54.524
Zero-Shot with None	13.928	52.058
Few-Shot with None	5.054	52.100
Finetuned	30.049	48.642
Finetuned with None	29.217	48.050

Table 6: Comparison between models and datasets of exact CWE accuracy (%).

C CWE Distribution in Data

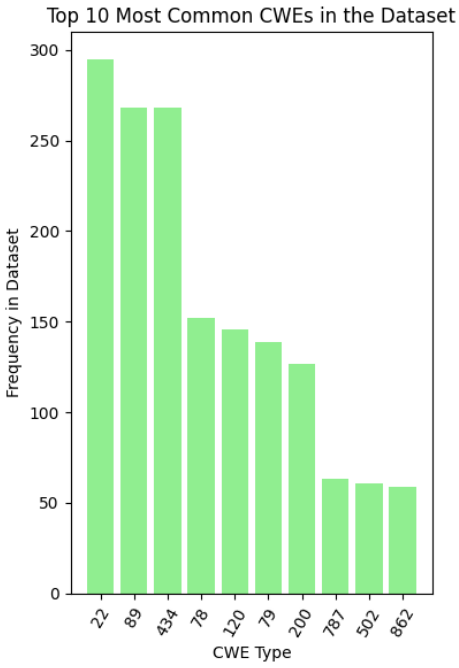

Figure 3: Top 10 most common CWEs in TitanVul

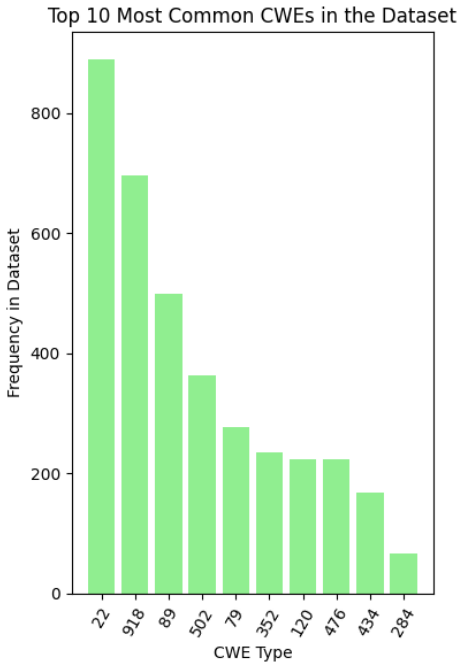

Figure 4: Top 10 most common CWEs in LLMVul

D CWE Accuracy and Frequency in Model Predictions

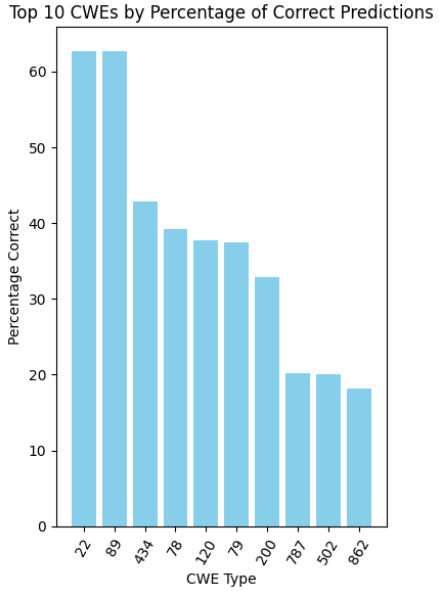

Figure 5: Accuracy of zero shot on TitanVul per CWE type

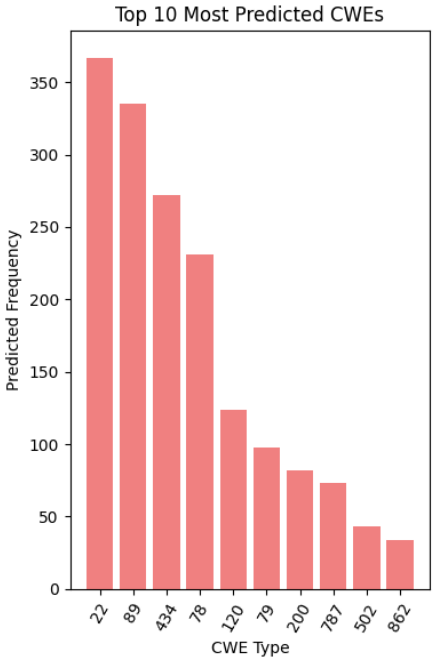

Figure 6: Count of most predicted CWEs with zero shot on TitanVul

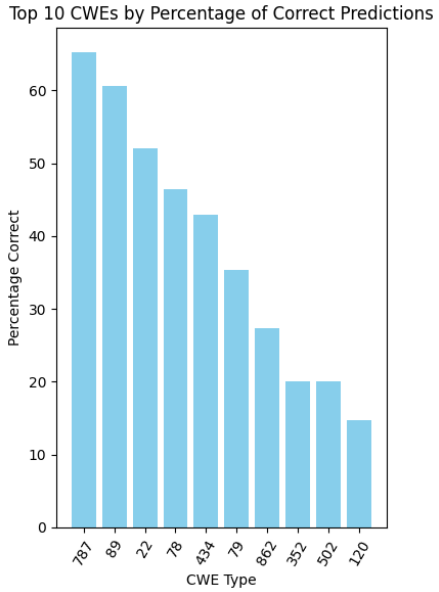

Figure 7: Accuracy of few shot on TitanVul per CWE type

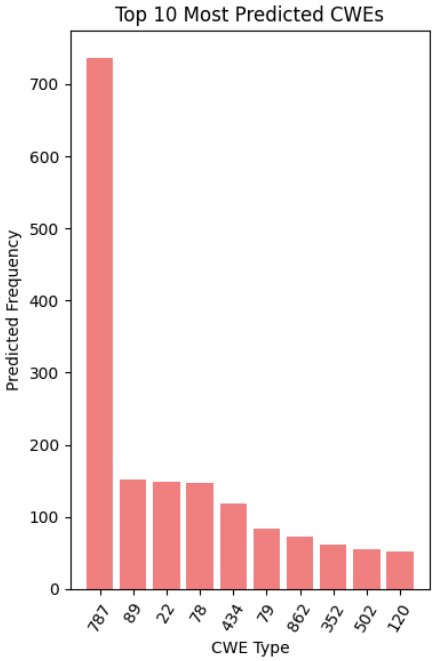

Figure 8: Count of most predicted CWEs with few shot on TitanVul

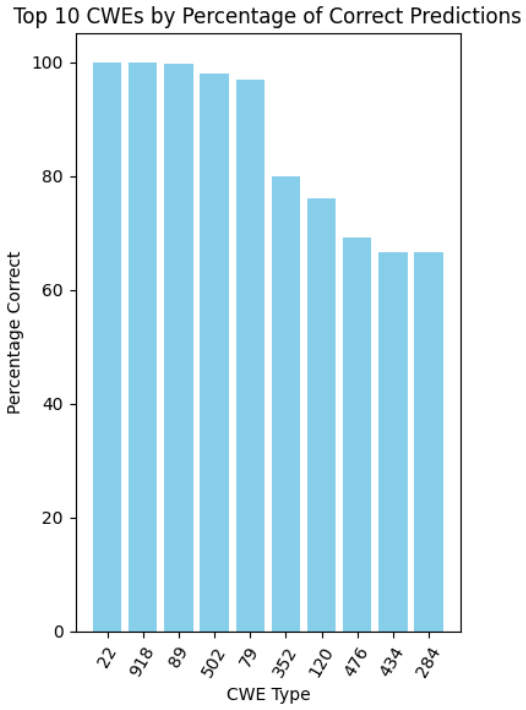

Figure 9: Accuracy of zero shot on LLMVul per CWE type

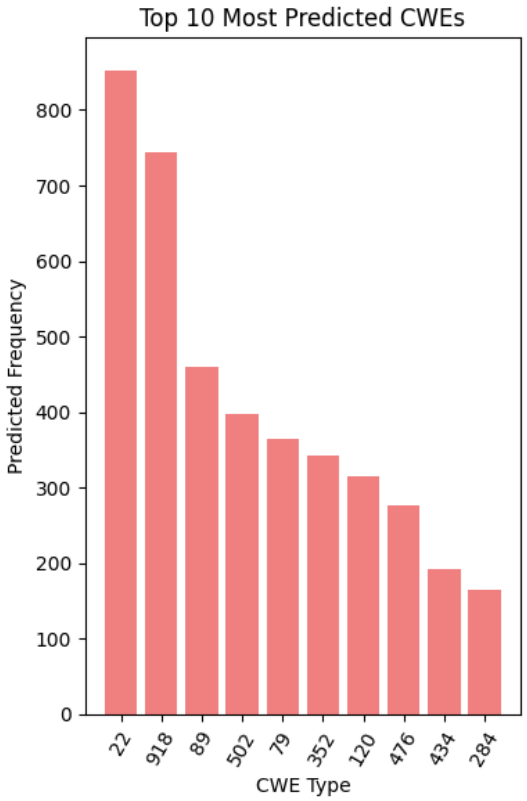

Figure 10: Count of most predicted CWEs with zero shot on LLMVul

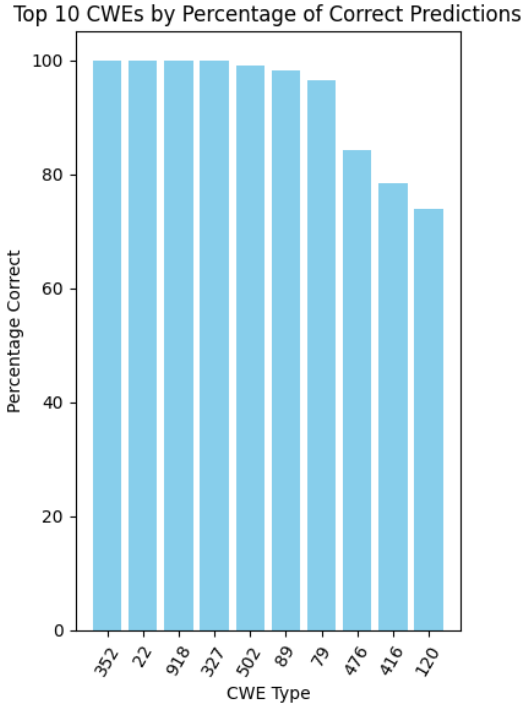

Figure 11: Accuracy of few shot on LLMVul per CWE type

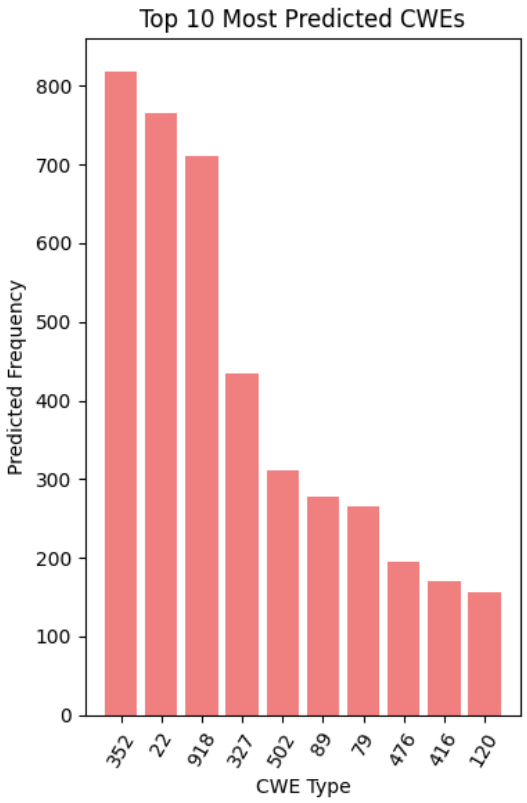

Figure 12: Count of most predicted CWEs with few shot on LLMVul

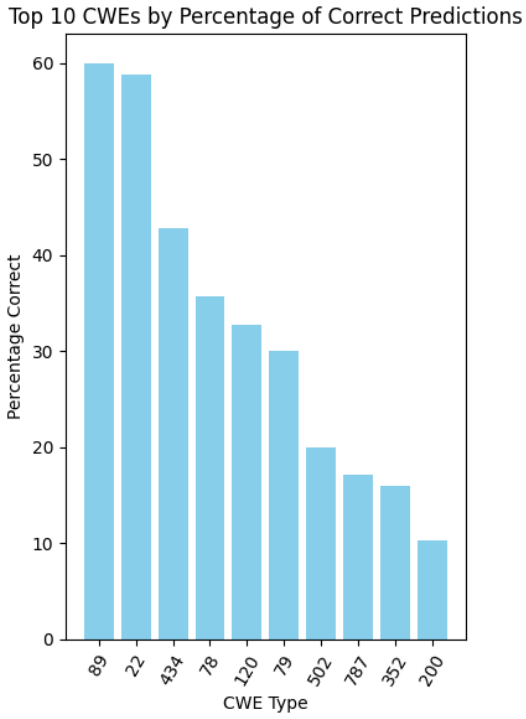

Figure 13: Accuracy of zero shot with choice of 'None' on TitanVul per CWE type

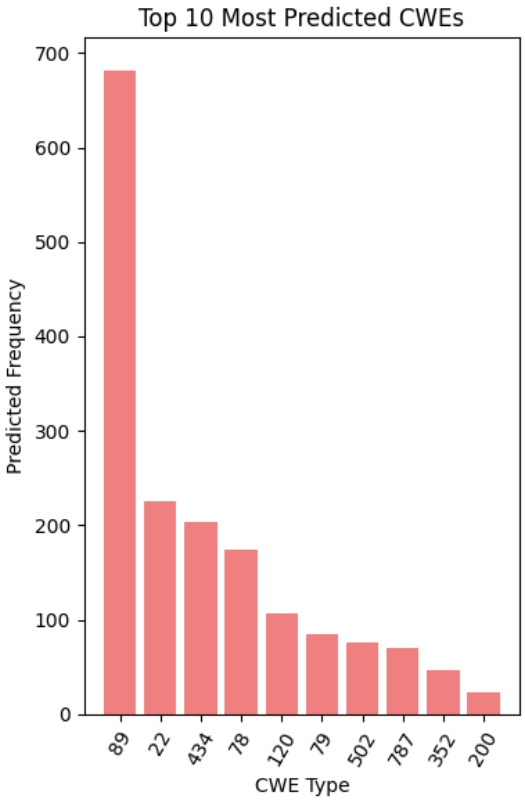

Figure 14: Count of most predicted CWEs with zero shot with choice of 'None' on TitanVul

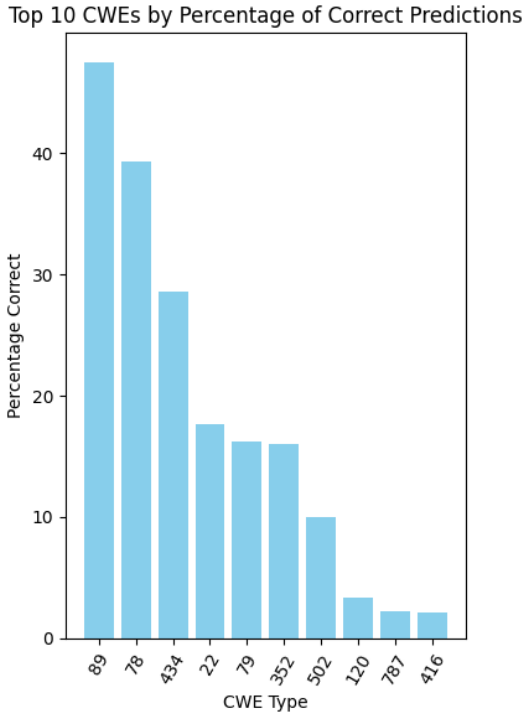

Figure 15: Accuracy of few shot choice of 'None' on TitanVul per CWE type

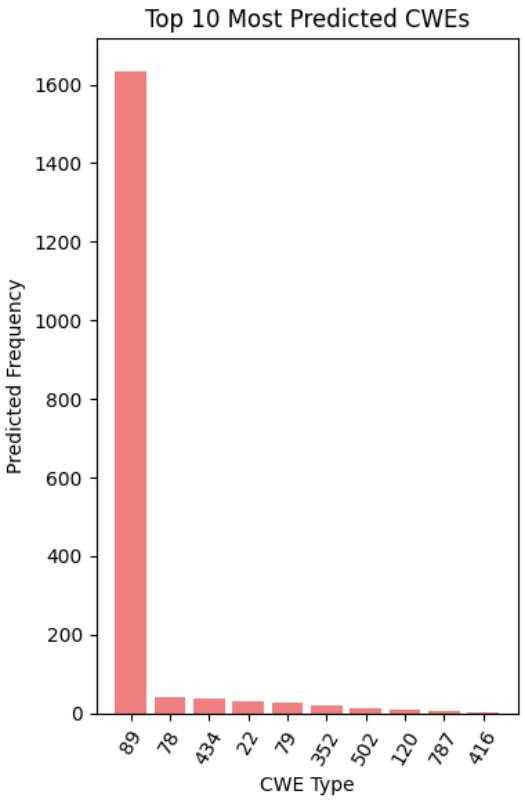

Figure 16: Count of most predicted CWEs with few shot choice of 'None' on TitanVul

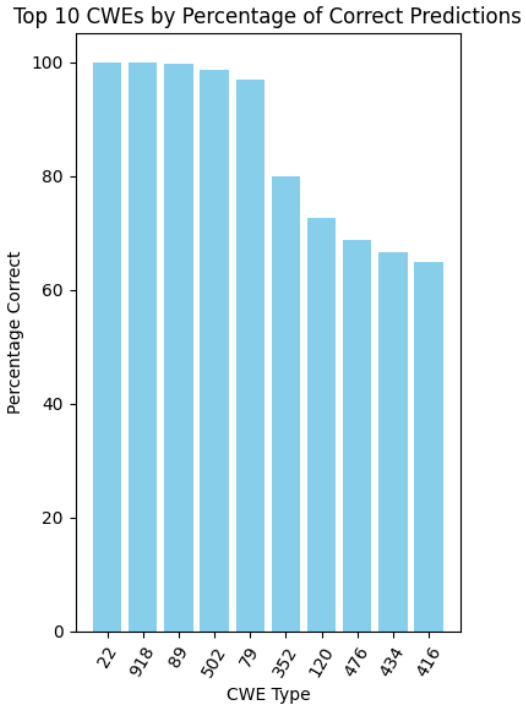

Figure 17: Accuracy of zero shot with choice of 'None' on LLMVul per CWE type

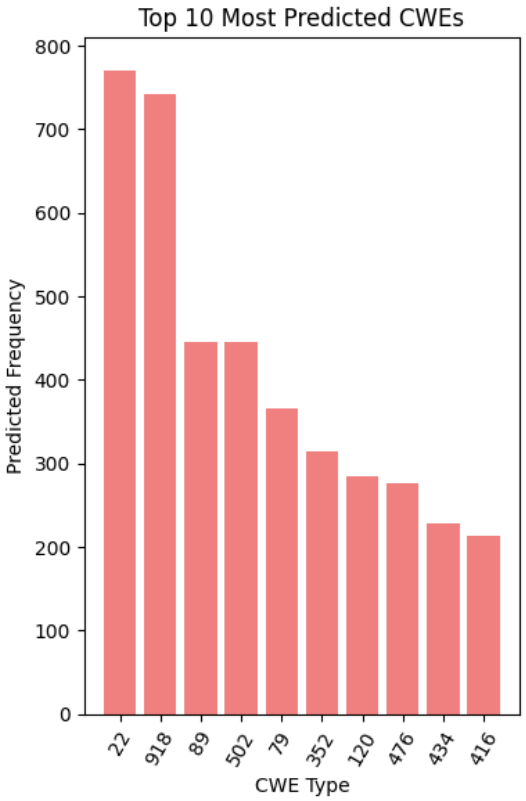

Figure 18: Count of most predicted CWEs with zero shot with choice of 'None' on LLMVul

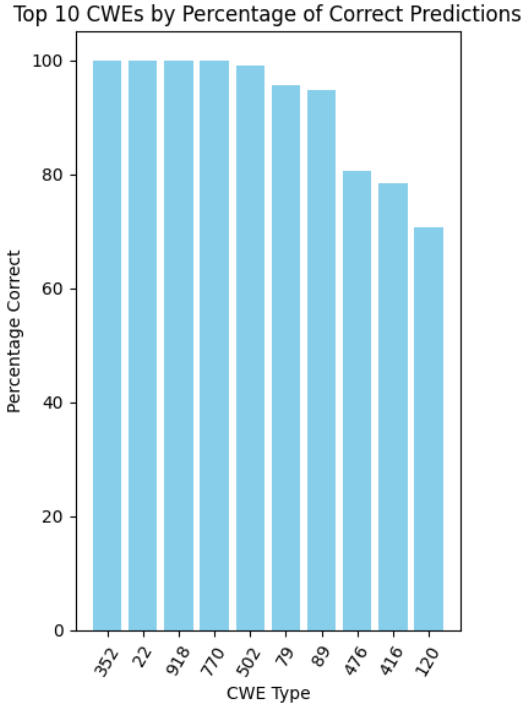

Figure 19: Accuracy of few shot choice of 'None' on LLMVul per CWE type

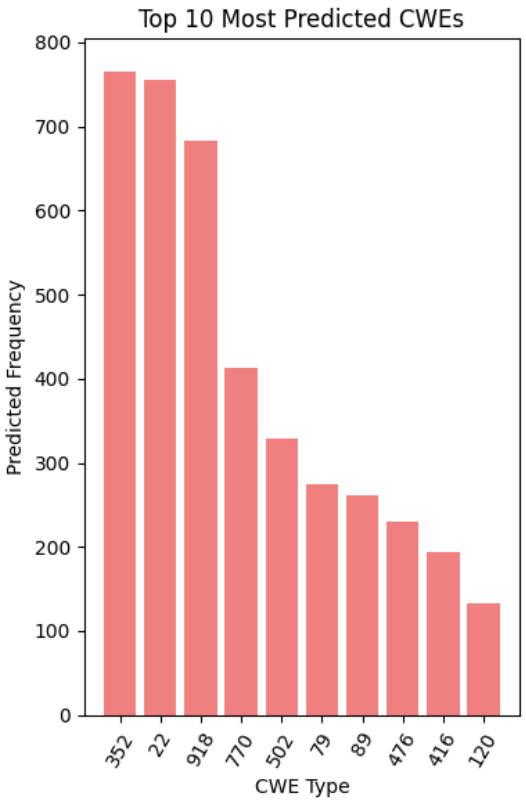

Figure 20: Count of most predicted CWEs with few shot choice of 'None' on LLMVul

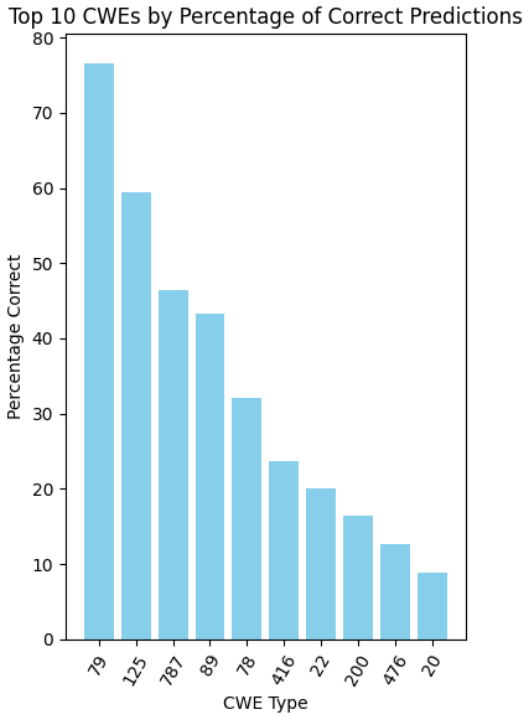

Figure 21: Accuracy of finetuned model on TitanVul per CWE type

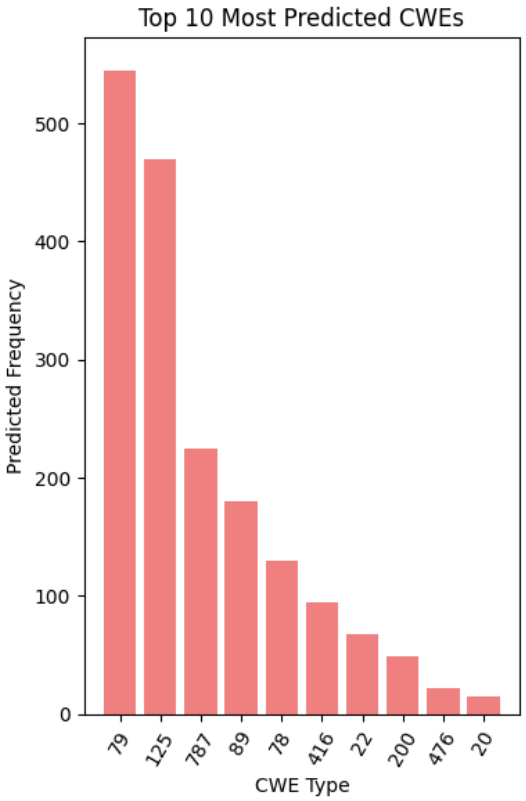

Figure 22: Count of most predicted CWEs with fine-tuned model on TitanVul

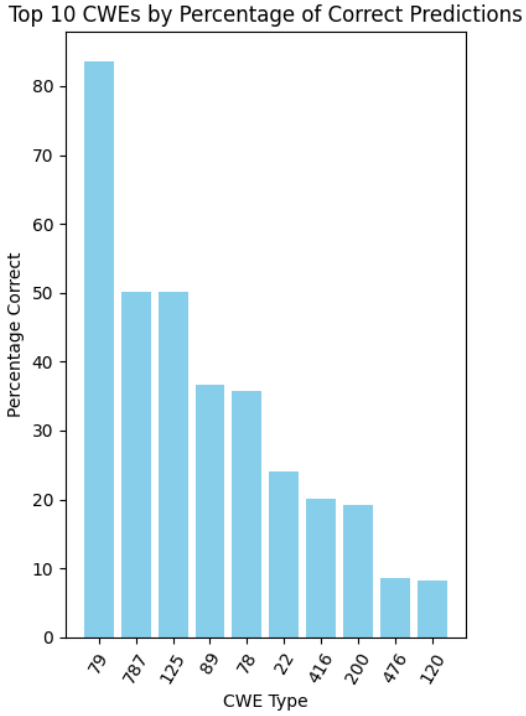

Figure 23: Accuracy of finetuned model choice of 'None' on TitanVul per CWE type

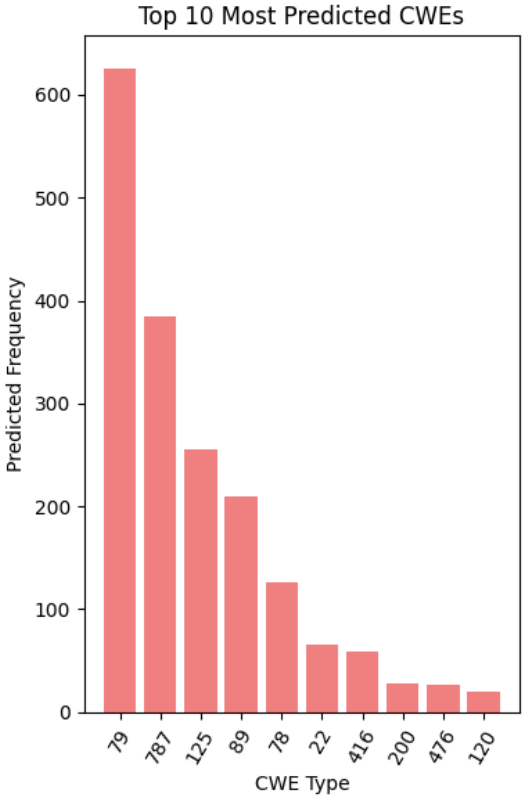

Figure 24: Count of most predicted CWEs with fine-tuned model choice of 'None' on TitanVul

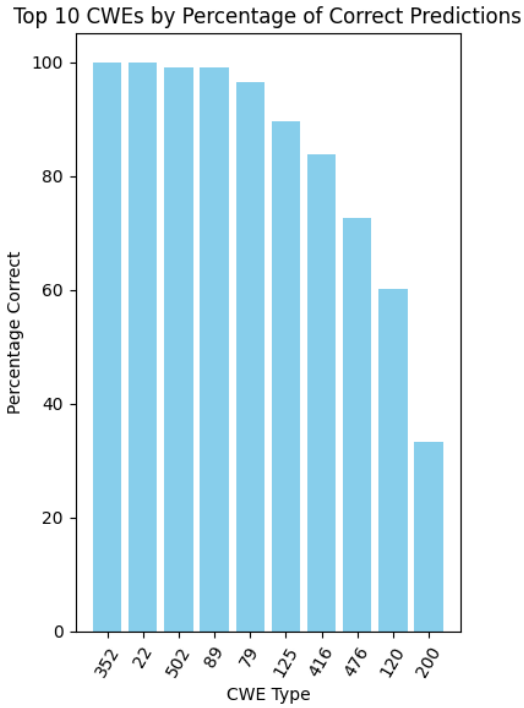

Figure 25: Accuracy of finetuned model on LLMVul per CWE type

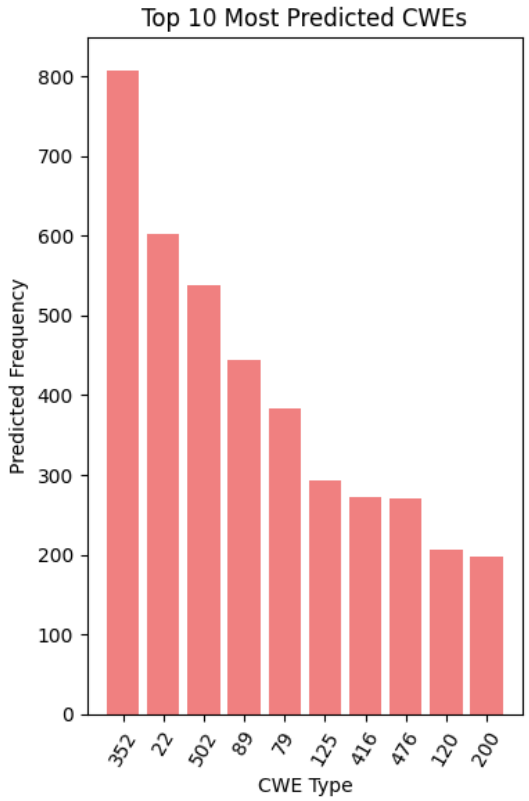

Figure 26: Count of most predicted CWEs with fine-tuned model on LLMVul

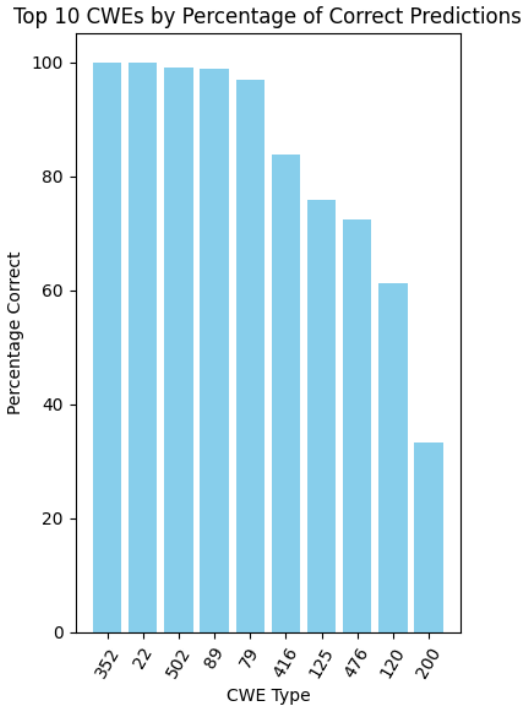

Figure 27: Accuracy of finetuned model choice of 'None' on LLMVul per CWE type

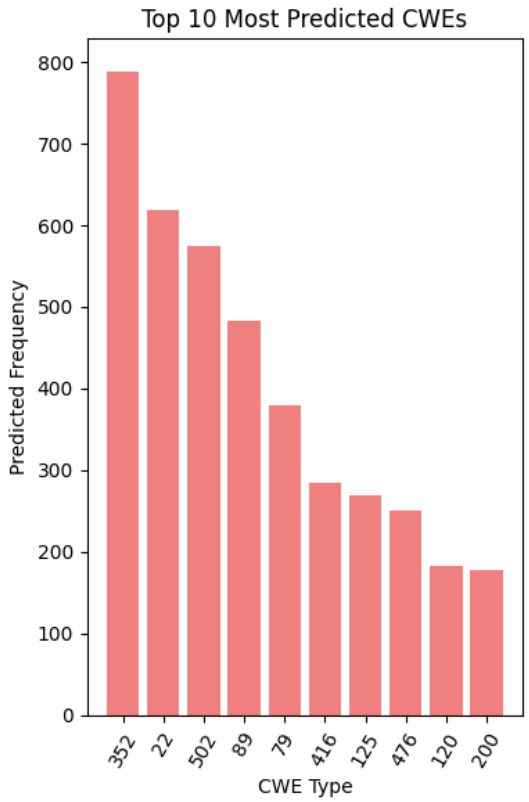

Figure 28: Count of most predicted CWEs with fine-tuned model choice of 'None' on LLMVul