# Securing the Creative Code: An Investigation into the Security Aspects of AI-Generated Code

by

Md. Nayeemul Hasan
22101622
Shoeb Mahmood
21101120
Jarin Islam
22101768
Mahbuba Zaman
21101152

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science and Engineering

Department of Computer Science and Engineering
Brac University
October 2025

# Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

**Student's Full Name & Signature:**

| | |
|---|---|
| Md. Nayeemul Hasan | Shoeb Mahmood |
| 22101622 | 21101120 |
| Jarin Islam | Mahbuba Zaman |
| 22101768 | 21101152 |

# Approval

The thesis titled "Securing the Creative Code: An Investigation into the Security Aspects of AI-Generated Code " submitted by

1. Md. Nayeemul Hasan (22101622)
2. Shoeb Mahmood (21101120)
3. Jarin Islam (22101768)
4. Mahbuba Zaman (21101152)

of Fall, 2024 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science in October, 2025.

**Examining Committee:**

Supervisor:
(Member)

Md. Aquib Azmain
Lecturer
Department of Computer Science and Engineering
BRAC University

Thesis Coordinator:
(Member)

Dr. Md. Golam Rabiul Alam, PhD
Professor
Department of Computer Science and Engineering
BRAC University

Head of Department:
(Chair)

Dr. Sadia Hamid Kazi, PhD
Chairperson and Associate Professor
Department of Computer Science and Engineering
BRAC University

# Abstract

AI-generated code has massive potential but this evolving technology also presents significant security concerns. This study investigates generative AI code and its security. It explores vulnerabilities inherent in AI-generated code and investigates methods to detect its vulnerability. This research analyzes the security implications of various Static Application Security Testing tools and generative AI models, identifies common vulnerabilities, and proposes practical solutions to audit and identify vulnerabilities. This study proposes an LLM that have been fine-tuned with a contextual dataset that is made up of SAST tool results. The proposed LLM is a 7B-Parameter fine-tuned Code Llama that has achieved a promising F1-Score of 0.42 and Recall of 0.72 in identifying vulnerabilities on unseen data. This study aims to contribute to the development of secure and creative coding practices in the era of AI-powered development.



**Keywords:** Software Security, Large Language Models, Code generation, Artificial Intelligence, Vulnerability Classification, Dataset, Cyber Security

# Table of Contents

# List of Figures

# List of Tables

# Nomenclature

The next list describes several symbols & abbreviation that will be later used within the body of the document

$AI$ Artificial Intelligence

$API$ Application Programming Interface

$APR$ Automated Program Repair

$AST$ Abstract syntax tree

$CERT$ Computer Emergency Response Team

$CI/CD$ Continuous Integration and Continuous Deployment

$CLI$ Command Line Interface

$CPS$ Cyber-Physical Systems

$CVSS$ Common Vulnerability Scoring System

$CWE$ Common Weakness Enumeration

$DAST$ Dynamic Application Security Testing

$ESBMC$ Efficient surface mount technology-Based Model Checker

$LLM$ Large Language Model

$LML$ Lean Manufacturing Language

$ML$ Machine Learning

$OWASP$ Open Worldwide Application Security Project

$SAST$ Static Application Security Testing

$SQL$ Structured Query Language

$XSS$ Cross-site Scripting

# Chapter 1

# Introduction

## 1.1 Background

Artificial intelligence (AI) and machine learning (ML) have become increasingly sophisticated, especially in terms of auto-generating computer code. Ten years ago, AI-powered code generation tools were little more than code-completion functions built into integrated development environments. Now, however, large language models (LLMs) and the generative approach have developed the capability of generating complex, near-complete programs with little to no human assistance. [39]

Neural network architectures allowed for a more generalized and powerful approach by training on large code datasets in multiple languages. These models could formalize programming constructs, coding style, and function patterns, enabling them to generate code that was often sufficiently coherent and syntactically correct. As organizations continue to deploy AI systems within their pipelines, the output of these systems (and any embedded instability) forms a crucial piece of their overall production infrastructure. [4]

While such AI-driven code generation improves efficiency and productivity, it also poses considerable security risks. These LLMs are trained on large code repositories, which may include insecure coding samples. Thus, these models could reproduce vulnerabilities like SQL injection bugs, buffer overflows, and poor cryptography if they are common in the training corpus. [15]

A parallel body of work is emerging that focuses on the security implications of AI-generated artifacts, in particularly software codes. Important considerations include model accountability, the shortcomings of automated testing and privacy and data protections. Tutorials demonstrate how to add secure datasets and filtration into training pipelines, including bias correction and fine-tuning on secure coding examples.

Liability and governance, responsible disclosure, and privacy and data protections are all regulatory and ethical implications of AI-generated code. When a security failure turns out to be a result of AI-generated code, who gets the blame, the developer, the model's provider or the organization implementing the code? [20]

Secure Coding Standards and Guidelines, Static and Dynamic Analysis, and Human-in-the-Loop Systems are just a few examples of mitigation strategies currently in the works. Working in strong security rulesets as training data for AIs and post-gen code reviews will allow researchers to catch and eliminate common exploits at the time of generation. Using traditional software testing tools along with specialized AI-based vulnerability detectors can reveal more nuanced defects prior to deployment.

This study's findings imply tangible near-term consequences for software engineering, cybersecurity, and technology policy, with valuable insights that both practitioners and researchers can adapt to help transition to AI-handled coding as a standard practice for software development in the future.

## 1.2 Motivation of the Study

The progress of AI and machine learning has been a massive boost in the software development process. Generative AI models are now able to produce fully working code snippets, refactor legacy systems and automate repetitive programming tasks. It is clear how these capabilities benefits faster development, cost savings, and potentially democratizing software authoring, even to non-experts. But this dependence on AI-generated code raises security concerns.

One of the motivations behind this study is the increasing number of evidences of AI models potentially generating code with vulnerabilities. Although they provide correct solutions, their training data can contain insecure code snippets. As a result, they may reproduce existing vulnerabilities, such as cross-site scripting vulnerabilities, improperly configured cryptographic secrets, and SQL injection risks. Many of these could remain undetected until deployed to production environments.

There are many studies regarding the security of the codes generated by AI. Automated testing, static analysis, and other traditional tools only detects the common vulnerabilities. The sophisticated and Zeor-day vulnerabilities could avoid being detected by these tools. This discrepancy in both research and practice illuminates the need to create and improve methods that specifically address the security evaluation of the generated code.

There are some software that integrate built-in security checking. But a single approach to detect vulnerable AI-generated code does not exist. [19]. An evolving study of security vulnerabilities in AI generated code is the needed since it is missing from both the practices of the industries as well as research in the academia. This research aims to provide a solution for software developers, security professionals, and AI platform developers.

If code generated by AI becomes insecure and untrustworthy, these tools may not be accepted by the consumers. Setting clear guidelines and improving detection systems will be key to maintain user trust and ensuring that AI's role in software development is sustainable in the long run.

## 1.3 Problem Statement

AI-assisted code creation speeds up the software development process and making coding more accessible. Unfortunately, dependence on massive training datasets frequently pulled from a plethora of public repositories that have not gone through rigorous vetting poses significant security challenges. The model could inherit vulnerabilities and they risk reproducing or propagating insecure coding patterns (e.g., SQL injections, unsafe cryptographic algorithms, or buffer overflows). There are tools to detect such vulnerabilities but because of industry consistently regulating human-generated code, they do not have the provisions for recognizing the specific dangers that arise from AI-powered development. This study answers the following research questions:

- **RQ1:** Is there a way to stop inheriting vulnerabilities?
- **RQ2:** Are the tools sophisticated enough to detect new exploits introduced by AI-generated code?
- **RQ3:** Should industries use more resources towards mitigating dangers that arise from AI-powered developement?
- **RQ4:** If any AI-generated code makes its way into production system, who should be held responsible?

## 1.4 Research Objective

The article below provides an overview if the approach to researching, and assessing the security concerns posed by AI-generated codes. The objective is to provide a foundation of practices for detecting and developing strategies to detect vulnerabilities.

1. Assess the Emergence of Vulnerabilities in AI-Generated Code
    - Examine the likelihood of AI models unwittingly learning and reproducing insecure coding patterns (such as those from public repositories).
    - Categorize common classes of vulnerabilities. For example, injection issues, buffer overflows, and logic flaws and study how these vulnerabilities propagate in generative outputs.
2. Evaluate Existing Security Evaluation Tools and Processes
    - Compare the capabilities and limitations of traditional static and dynamic analysis tools for detecting vulnerabilities unique to AI-generated code.
    - Examine next-gen AI-based security scanners and evaluate them against legacy solutions to assess their capability to discover obscure or novel exploit vectors.
3. Systematic Approach For Vulnerability Detection

- Develop a validated framework of human-in-the-loop, automated scanning, and model-specific checks that can minimize the inadvertent infusion of exploitable code.
- Outline procedures for secure coding pipelines, data filtering, fine-tuned secure coding examples, and testing protocols after code generation.

4. Analysis of Real-World Applications and Recommendations

   - Systematically measure prevalence of security bugs in AI-generated software and evaluate the efficacy of proposed efforts to prevent them with empirical testing of real software deployed in the real world.
   - Recommend best practices for software developers, security personnel, and AI service providers,addressing secure coding standards, liability concerns, and responsible disclosure policies.

## 1.5 Methodology in Brief

**Data Collection**
Specifically, the study can involve harvesting code samples generated by AI tools such as ChatGPT, Claude, Deepseek and many other code-generation models. These samples cover a many programming tasks, such as: web development, cryptographic functions, API integrations, and database queries etc. This data will be collected from public datasets, and prior research papers.

**Code Analysis**
Both static and dynamic methods will be utilized to analyze the collected code samples in order to identify security vulnerabilities. Static analysis tools like SonarQube, Bandit, and custom scripts will identify common issues like SQL injection, XSS vulnerabilities, hard-coded secrets, and error handling. Static analysis can include simulating runtime to find flaws such as buffer overflows, use of insecure API systems and poor validation of input. Besides automated techniques, qualitative examinations will be performed to determine logical vulnerabilities and bad coding practices that can not be found by tools.

**Model Fine-tuning**
The study will distinguish between security-aware and security-agnostic training by fine-tuning an existing AI code-generation model. Fine-tuning is when you take a pre-trained model (like one trained on lots of human-written text) and adjust the later layers with a dataset that is focused on secure coding practices or with code that is labelled either insecure or secure. This process would enable the evaluation of whether the fine-tuned model generates output that is more secure than that from the non-fine-tuned version.

**Evaluation**
The fine-tuned model is to be evaluated against non-fine-tuning models on various criteria (e.g., when compared to non-fine-tuning models, what is the level of security, function and performance of fine-tuned models?). They will utilize metrics including, for example, vulnerabilities detected per 100 lines of code, vulnerabilities severity

(CVSS-standardized), and compliance with secure coding guidelines (OWASP Top 10 or CERT).

**Iterative Testing**
In iterative testing, the fine-tuned model will be tested with new prompts and datasets multiple times for tuning its outputs. Feedback loops will be used to identify weaknesses and will be added to the training dataset to improve the performance of the model. The iterative process also assesses the model's overall performance across lessons involving security contexts.

## 1.6 Scopes and Challenges

**Scopes of the Research**

The study's potential lies in its ability to assess the security of AI-generated code by using AI itself to detect vulnerabilities. AI tools proposed by this research could become a reference that many developers might need. This reduces the amount of time spent on identifying vulnerabilities and allows developers to spend more time on feature development and optimization.

The findings of this study can help provide a framework for developing best practices for the use of AI-generated code in domains such as, healthcare, finance, and cybersecurity. These findings could also used for Automated Program Repair for programs, static analysis augmentation, as well as secure coding education. Explorations into further engineering approaches, such as formal verification and utilizing vulnerability-aware training datasets, could lead to the development of new tools and techniques.

**Challenges**

Models are trained on publicly available code repositories that contain good quality code but also vulnerable code. It's still a big problem to make sure all training datasets being used have been cleaned for security. Also, the bias on datasets can also reproduce certain patterns of vulnerabilities. Analysis tools sometimes miss complex, logical vulnerabilities in AI-generated code. There are also Zero-day vulnerabilities which are extremely difficult to detect by the tools that are available now. Developing tools to detect Zero-day vulnerabilities is equally difficult.

There needs to be a balance between automation and human oversight. AI systems could give a false sense of security. Thus, there is a risk of developers becoming too dependent on processing through these systems without manual review processes or comprehensive security protocols. Furthermore, there are also computational resource constraints. Techniques such as formal verification or other advanced methods for vulnerability detection are computationally intensive and difficult to scale to real-world scenarios or datasets. The existing methods of fine-tuning LLMs for cybersecurity tasks require a significant amount of resources, which may be difficult for many research teams to obtain. Cybersecurity is a fast-moving space,

and attackers are always reinventing themselves to breach the latest cyber defense. Models trained on historical vulnerabilities could have difficulty adjusting to the vectors of new attacks. Therefore, we need a balanced approach that incorporates AI powered threat detections and human oversight to actively train models to adapt to evolving threats.

# Chapter 2

# Literature Review

## 2.1 Preliminaries

This phase of research sets the framework of concepts, tools, and techniques foundational for all future research on the security concerns of AI-generated code. This section describes the main technologies, frameworks, and terminologies involved, creating the context for the study.

The neural network-based architecture, referred to as Large Language Models (LLM), are trained on large chunks of text and code [28]. Such systems produce text and software code in a human-like manner in response to input text prompts. By training on data, they can output snippets and functions, or even entire programs, based on their training data, increasing productivity but raising the specter of replicating insecure patterns in the datasets.

The Common Weakness Enumeration (CWE) defines a software vulnerability by categorizing vulnerabilities into classes of similar properties [3], such as Cross-Site Scripting (XSS), SQL injections, and other flaws. However, training on unsanitized code repositories amplifies those vulnerabilities, and emphasizes the need for hardening and safety mechanisms.

For identifying vulnerabilities in AI-generated code, techniques such as Bandit and CodeQL rely on static analysis tools [36], while formal verification methods such as symbolic execution and model checking provide mathematical guarantees about code correctness and security despite computationally expensive. Both of these approaches are equally important in assessing the security of AI-generated code.

Techniques from natural language understanding are used to identify and correct vulnerabilities in code. It helps LLMs to generate secure code, introducing fewer vulnerabilities. Such techniques are being used in increasing numbers for automated program repair (APR).

Cyber-Physical Systems (CPS) systems vulnerabilities testing in AI-generated code is performed, using curated datasets such as FormAI [10] and CodeSecEval [33], which provide insights into the behavior and the risks posed by these systems.

## 2.2 Review of Existing Research

### 2.2.1 Security of AI generated Codes

In a systematic study, it was analyzed that AI generated code across common programming languages and shows that a significant portion of the generated code contain different vulnerabilities such as SQL Injection, insecure cryptographic algorithms and Cross-Site Scripting (XSS). It stresses the importance of training data in relation to code security, since, generative AI tools reuse patterns from their training datasets, which typically contain publicly available, unvetted and insecure code. The reliance on and replication of these insecure coding techniques is then performed, resulting in both the replication and, in some cases, amplification of insecure coding practice. [6]

The authors raise authenticity challenges, since a large part of the code generated very closely follows or blatantly copy-pastes patterns seen in publicly available repositories, creating ethical and copyright concerns. The lack of detailed transparency over the origin of AI training data makes determining whether AI-generated code is original challenging. [6]

In a research study where authors analyze 3,854 programs synthesized by ChatGPT and Bard through 133 unique Google Code Jam competitive programming problems. It finds patterns of security vulnerabilities in AI-generated code. It demonstrates how certain types of vulnerabilities manifest across different languages and problem contexts. More than 50% of AI generated code is insecure, including 100% of code generated in C. ChatGPT and Bard scored better when asked to write in Java and Python, but flaws like these remained widespread, especially given the lack of proper input sanitization, memory safety, and other measures. [14]

These models showed limitations when asked to evaluate and fix their own generated code. Including inability to spot complex or context-dependent vulnerabilities [13]. And across programming languages, performance was consistently insecure, with memory management vulnerabilities being a common issue. When prompted to find vulnerabilities in code generated by ChatGPT, Bard demonstrated higher rates of detection and correction. [14]

### 2.2.2 Impact of Prompt Engineering

Some studies explore the utility of LLMs like GPT-4 and Codex for repairing defects identified during code review. The work how critical prompt design is in harnessing LLMs to formulate correct and safe fixes for the discovered vulnerabilities. In this work, the authors analyze the performance of LLMs on a set of common code-review quantum quality defects on a variety of scenarios to also reflect on the abilities and limitations of LLMs and their potential support of software development workflows. [12]

The need for fast assessment is highlighted, as it will ultimately improve the quality and also the productivity when it comes to Lean Manufacturing Language (LML)

defect solution. The research that the quality and specificity of prompts greatly affect the accuracy and relevance of code fixes generated by LLMs. The authors compared various Prompt Strategies: Context-aware prompts explicitly describing defects while providing code snippets relevant to the task can guide sample selection and yield improvement over vague or underspecified prompts. [12]

LLMs were evaluated across a range of categories for defects frequently detected as part of code reviews (e.g., Logic Errors, Style Issues, Security Flaws). The models had different performances. Which means addressing style and readability was fixed but failed when tasked with correcting logic errors or ensuring if vulnerabilities were eliminated. [12]

In software engineering task, LLMs are being used to automate repetitive tasks. Although LLMs have the potential to improve human productivity under human oversight, the paper notes challenges such as dataset bias, scalability, and reliability that should be solved before such tools can be directly used into the industry workflows. By addressing prompt engineering and some other methods as part of an iterative process, developers are offered practical suggestions to unlock the full potential of generative AI in the software development lifecycle. [35]

EvalGPTFix is a dataset engineered to assess the bug-fixing capabilities of ChatGPT. It was done without being influenced by prior training knowledge. Using this dataset, ChatGPT was able to debug 109 out of the 151 buggy programs, and achieved even higher success (143 of the bugs fixed) when enhanced prompts and interactive dialogues were used. This shows the role of well-designed prompts and iterative interactions in LLM performance. [35]

### 2.2.3 Generative AI in Cyber Security

Despite the clear benefit that LLMs provide in terms of convenience and efficiency, there are still significant limitations. Most notably their inability to comprehend vulnerabilities that relate to multiple files or are dependent on complex contexts. The authors note that zero-shot techniques are very positive, but insufficient in real-world situations and that the models will sometimes introduce new vulnerabilities as they try to repair them. Additionally, the dependency on prompt quality also makes it particularly hard to scale. These limitations can be solved by expanding the datasets and fine-tuning the models with security-specific datasets so that LLMs understand and comprehend vulnerabilities that have their own nuances. [9]

The dual-use nature of generative AI is very risky. The models perform well in language based tasks but they are still not yet scalable or large-scale workflow capable. The study highlights the gaps when it come to domain specific tools. There are also insufficient ways to validate them as well. Furthermore, while the paper does discuss about the risks, it does not mention how to guard against them. [31]

GPT-4 and Codex have shown promise in helping find hidden security problems [41], generate secure code, and automate test case generation for security-critical paths.

During software lifecycle, these models provide support in architectural risk analysis and operation through monitoring runtime behaviour during deployment. The capacity of generative AI to conduct cross-language analysis and automate repetitive tasks with minimal retraining saves a lot of time, especially for large projects. [21]

Generative AI is a double-edged sword for CPS security. It has both potential and risks. Potential implementations for Cyber Security using generative AI models like ChatGPT: Automating threat detection and response with generative AI can help identify and respond to potential threats in real time, integrating tools to monitor systems. They can also be used to simulate attack scenarios, improve predictive maintenance models, and increase resilience by providing real-time information on security incidents. [30]

Output from Generative AI models can also be malicious in nature (e.g., phishing scripts, malware) and can be subject to adversarial attacks, such as prompt injection. However, the limited context of these models is why they are less effective in solving very large and interconnected CPS environments, and their over-reliance on AI-driven automation might lead to general complacency in humans oversight of complex systems. In addition, empirical validation and practical implementation examples are not available in the paper and it making the applicability of the finding weak. Multi-modal AI models are discussed only briefly and lack analysis, a missed opportunity as they may be used to model combined data streams such as textual (e.g., network logs) and network (e.g., control systems) data streams for significant improvement in CPS security. [30]

### 2.2.4 Usage of Public Repositories for Model-Training

Some studies apply static analysis to 733 pieces of code generated by Copilot on open-source GitHub projects, using tools like CodeQL. Based on the CWE, the analysis provides a breakdown of the classes and frequencies of security weaknesses. Security vulnerabilities are indeed an issue with Copilot-generated snippets, with 30% of snippets found to contain at least one security vulnerability across 43 CWE categories such as insufficiently random values, improper control of code generation leading to possibilities of code injection and XSS. [24]

Copilot regularly adds security vulnerabilities that are less likely found in code created by humans. Most of these vulnerabilities are a byproduct of Copilot mirroring insecure patterns found in its training datasets. Post-generation fix efficacy was further enhanced by using Copilot Chat with static analysis tool security warnings. Guided by warning messages, Copilot corrected 55.5% of all identified vulnerabilities, indicating a path to secure coding in a supervised workflow. [24]

Using CodeQL to perform static analysis to locate vulnerabilities and classify them under the CWE framework. ChatGPT-generated code was free from vulnerabilities in 20% less CWE categories than StackOverflow snippets, with the first containing vulnerabilities across nine CWE categories and the second one across 22. 25% of vulnerabilities were specific to the ChatGPT-generated code, showing different

patterns of problem. Issues such as broken cryptographic practices affected ChatGPT notably, while snippets from StackOverflow were more likely to suffer from unsafe resource handling and the use of obsolete coding practices. [25]

### 2.2.5 Specialized Datasets to Detect Vulnerability

The FormAI Dataset is a dataset for studying AI-Generated C Programs in Software Security. It employs formal verification methods such as symbolic execution and bounded model checking to establish the functional correctness and security of AI-generated source code. The researchers use tools such as ESBMC to identify potential bugs. Most notably, AI-generated programs failed formal verification 76% of the time, with common bugs being memory safety issues, logical errors, and unsafe patterns. GPT-3 and other generative AI models create human-readable text from scratch. CodeT5 and Codex, are benchmarked against the dataset but tend to fail to maintain correctness and security logic, leading to a high failure rate during formal verification. [10]

There are 180 annotated samples on SecuCoGen covering 21 major vulnerability types of the "2023 CWE Top 25 Most Dangerous Software Weaknesses". This dataset is benchmark of fine-tunied LLMs for enhanced security awareness. Using SecuCoGen, the authors tested models (e.g., GPT-4, CodeT5, and Codex) in three tasks: secure code generation, vulnerability repair, and vulnerability classification. The implications of these findings illustrate that while these models can produce functional code, security concerns are often neglected during this generation process. It results in insecure products and inconsistent repair performance. [11]

Based on the CWE framework, this report presents an analysis of these vulnerabilities and displays those of common occurrence, including XSS, SQL Injection, and broken or risky cryptographic algorithms, etc. The study also points to the effect of training data in proliferating outdated or insecure coding practices. [34]

### 2.2.6 A dedicated tool for security assessment

DeVAIC is a tool for assessing the security of AI-generated Python code. It uses pattern matching techniques that are currently based on CWE categories to analyse incomplete code snippets. It helps filling a crucial gap left by traditional static analyzers. By identifying 35 CWE categories, including high-risk issues such as deserialization flaws and improper input validation, DeVAIC achieved a 94% accuracy rate and in the paper, it references reduced false-positive count compared to programs such as CodeQL and Bandit. It has an average processing time of only 0.14 seconds per snippet, making it lightweight enough for use in real-time or at a large scale, such as in CI/CD pipelines. [37]

A major advantage of DeVAIC is its ability to focus on incomplete code. Instead of using heavy AST modeling, DeVAIC comes with built-in detection rules that make it easy to use without too much configuration. The tool's assessment of real-world

Python code are produced by popular AI models like ChatGPT. CodeT5 and GPT-4. The study's focused on addressing security challenges that arise from AI-generated code. [37]

### 2.2.7 Developers on Generative AI

Reddit, StackOverflow, and GitHub forums are one way to find incidents regarding security vulnerabilities associated with AI-generated Code. This raises serious questions regarding the security aspects of AI-generated code, such as replicating insecure patterns from training data, potentially creating insecure code, and missing common mistakes by inexperienced users. [29]

There are also concerns about ethics and intellectual property. There are questions on whether AI-compiled code violates licenses of the datasets it was trained on. The report provides some concrete suggestions for generative AI tools such as being able to provide real-time feedback, clearer disclaimers on risks for user understanding and transparency on the code generation process. [29]

Generative AI tools are helping the software development process but there is also an urgent need to understand how the tools influence developers. Utilizing real-world conversations, some studies reflect genuine developer opinions, providing practical takeaways that can be implemented immediately to improve these tools. However, the paper's emphasis on public forums is limiting, since it may miss insights from either private or formal evaluations. Furthermore, the lack of some quantitative metrics limits the comprehensiveness of the findings. [29]

Some studies also discuss the policy and governance dilemmas, such as making clear who is accountable when an AI tool introduces a vulnerability, what responsible practices should be in place when using generative AI to develop software, and what transparency should exist around AI model training data and decision making. [26]

## 2.3 Summary of Key Findings

ChatGPT outperformed the models such as CodeT5 and PLBART and demonstrates effectiveness in Automated Program Repair (APR). It was able to repair 109 buggy programs from a pool of 151 applications through a single context-free prompt, with the number increasing to 143 with interactive feedback. This work emphasizes the need for prompt engineering and provides evidence that well-structured prompts improve ChatGPT's abilities to fix bugs. ChatGPT improved prediction accuracy by 27.5% over CodeT5 and by 62.4% over PLBART. On the other hand, there are concerns about data leakage in publicly available datasets such as QuixBugs and Defects4J. It raises questions regarding fairness in evaluating how well an AI will perform. Despite the concerns, ChatGPT's flexibility demonstrates its promise for use in real-world software engineering applications, especially when accompanied by human developers to provide oversight. [35]

AI-powered tools like GitHub Copilot are becoming popular among developers for automating repetitive tasks, generate reusable code, and help with the implementation of complex logic. However, there are concerns that AI-generated code might replicate insecure patterns found in its training data. For new developers, this is a challenge, as they may find it difficult to differentiate between secure and flawed AI-generated code. Consequently, the developer community has a critical role in defining best practices, mitigating AI-generated problems, and voice their opinions about the ethical considerations surrounding generative AI tools. [29]

Even the detection of security vulnerabilities saw an advancement with AI-based tools such as DeVAIC that has an accuracy of 94% in terms of vulnerability discovery through 35 CWE categories, including all high-risk OWASP Top 10 attacks. While traditional static analysis tools are built for complete applications in constructing ASTs, DeVAIC can analyze code snippets (even around AI-generated code). In comparison to existing tools such as CodeQL, Bandit, and Semgrep, DeVAIC excelled at accurately detecting vulnerabilities when querying incomplete code snippets. However, being based on pattern-matching techniques it is not well suitable for detecting novel or very complex vulnerabilities. [37]

Examples of Secure Code Generation and Repair in the domain of secure code generation and repair, AI-powered models like GPT-4 and Codex have demonstrated effectiveness in correcting simplistic security flaws without the need for further tuning [11]. However, they are poorly-suited to complex or context-dependent security problems, especially those which cross file or module boundaries. This limitation highlights the need for human validation because LLM-generated fixes may be susceptible to itself, inadvertently introducing new vulnerabilities. SecuCoGen is also another dataset for benchmarking secure code generation, repair, and classification tasks. [9]

While generative AI brings many advantages, it also poses cybersecurity risks. The misuse of AI for generating such content can be adjusted to launch highly involved and comprehensive phishing assaults, produce malwares that are harder to understand, and sidestep security protocols via adversarial prompts [31]. Even though AI can help in automating software security, the bias and inaccuracies in generated outputs will require human involvement to be really effective in the long run [21]. Artificial Intelligence not only helps organizations identify potential threats but also plays a significant role in Cyber-Physical Systems (CPS) security. AI can create real-time threat models, monitors network anomalies for unusual user behaviors, automates vulnerability scanning and patch management etc. Nonetheless, the incorporation of AI into large-scale CPS is still a challenging task due to the complexity of these systems.[30]

Generated code from AI mostly reproduces established security flaws such as CWE-79 (Cross-Site Scripting), CWE-89 (SQL Injection), CWE-327 (Weak Cryptography) [34]. In a comparison of code generated by ChatGPT and samples from StackOverflow, it is [25] found that ChatGPT's code has 20% fewer vulnerabilities but is also vulnerable to security weaknesses. Static analysis using tools like CodeQL shows that these vulnerabilities are widespread in AI-generated code. If ChatGPT

or Bard are prompted to identify and patch vulnerabilities that depend on multiple interacting files, they both struggle to complete that task [14]. This showcases a significant limitation of AI in understanding such intricate vulnerabilities in logic. Thus, human oversight is essential in making sure AI-generated code is secure. [6]

AI security capabilities have achieved great results in these areas of social networks. When Copilot Chat was paired with static analysis tools like CodeQL, it effectively resolved 55.5% of discovered vulnerabilities [24], highlighting the potential of hybrid AI-human approaches to software security. Generating secure and functionally correct C code is a multi-step process with constraints on object lifetime that few AI models have demonstrated an ability to tackle, which is why low-level programming is believed to be a particularly difficult domain for generative models. The newly released dataset FormAI [10], consisting of 112,000 real-world, AI-generated C programs, demonstrates the common vulnerabilities of AI-generated outputs and serves as a strong baseline for follow-up studies. Additionally, research on prompt effectiveness indicates that the use of context-rich prompts improves repair accuracy significantly over minimal or error-focused prompts [12]. As we move forward, methods for reducing vulnerability propagation through model fine-tuning, real-time security feedback into models, and hybrid AI-security approaches must increase AI-generated code reliability. [26]

# Chapter 3

# Requirements, Impacts and Constraints

This chapter details the technical specifications, broader impacts, and practical constraints of the research project. It provides an overview of the resources required, the societal and ethical impacts of the work, a plan for managing the project and economic analysis.

## 3.1 Final Specifications and Requirements

The main idea of this research is to fine-tune a Large Language Model which is computationally very expensive. To run a 7B-Parameter model, 12GB of VRAM, and 16 to 32GB of memory is recommended. The models utilize CUDA which is developed and available to use in NVIDIA GPUs [38]. This study was conducted on a system equipped with Nvidia RTX 3080 with 10GB of VRAM, an AMD Ryzen 7700 CPU and 32GB ram. Even with this hardware, we had to use QLoRA to keep the VRAM from overloading.

The research was conducted in a multi-phased methodology that required a verifiable baseline, systematic analysis and iterative fine-tuning. Starting with a high quality dataset, a systematic SAST tool analysis was required to build the ground truth for model evaluation. Then through iterative testing a well performing model was achieved. The implementations relied on open source software. The core language was Python, Python 3.10 environment along with Pytorch and Hugging Face ecosystem was utilized. For analysis tools, Cppcheck, Flawfinder, Bandit, Semgrep, Snyk, Pylint and Vulnhuntr were used. For access to a generic public LLM, Gemini API was used to access Gemini 2.5 Pro and Gemini 2.0 Flash.

## 3.2 Societal Impact

The usage of AI in development or in general have proven societal impacts. If AI-generated code is deployed in production or in critical systems such as in medical field, vulnerable software can lead to system failure. It could cause harm or loss of life. The technology also creates legal gray area regarding liability. If a security breach occurs due to a flaw in AI-generated code, who should be help responsible? The

developer who built the code? The organization who deployed it? Or the company that created the AI model? An over-reliance on AI-generated code could lead to danger, skill dispersion and security issues [32].

## 3.3 Environmental Impact

Training LLMs is an energy intensive process. LLMs require powerful GPUs to run and this contributes greatly to carbon footprint. Our experiments, even with a relatively small 7B parameter models on a single machine, needed many hours of high intensity computation. A single training on a model alone required 33 hours and a single evaluation took well over 28 hours [23].

## 3.4 Ethical Issues

LLMs are trained on public code from repositories such as GitHub. This study confirmed that the code are often insecure. The models learn from these code and reproduce vulnerabilities and may even create more vulnerabilities. This creates an ethical issue as the tools may guide less experienced developers to build flawed and vulnerable systems [40].

The same AI that can be used to detect vulnerabilities, can also be used to develop malware and discover new exploits the same way. This is a dual-use nature that is an inherent ethical challenge in AI for cybersecurity [40].

## 3.5 Project Management Plan

This academic study was conducted by four students with an estimated 12-month timeline. The four-person team divided tasks to run in parallel where possible. For example, if one member focused on data processing another member configured and monitored model training progress. The single GPU machine was the constraint and the schedules were carefully managed to maximize productivity. Our research timeline is as follows:

- **Month 1:** Literature review and research proposal finalization.
- **Month 2:** Environment setup, tools installation and dataset acquisition.
- **Month 3-4:** Execution of SAST tools, recreating FormAI labeling, Python translation and vulnerability inheritance investigation.
- **Month 5:** Training dataset preparation, model evaluation ground truth labeling.
- **Month 6-7:** Model preparation, training with Training Dataset 1 and evaluation.
- **Month 8:** Optimizations of the models, re-evaluation of models.
- **Month 9-10:** Fine-tuning models, re-training and re-evaluation.

- **Month 11-12:** Analysis through API, Thesis writing, result compilation and validation, Final submission.

## 3.6 Risk Management

The project had faced several risks which were managed with different methods and options. The computational resource constraint such as VRAM limitation was one of the risk factors. This was mitigated by implementing QLoRA for 4 bit quantization. The training dataset was also reduced to fit in the VRAM. The initial fine-tuning approach failed because of dataset not containing secure code samples, and the existing ones being snippets rather than entire code. This was fixed by using token analysis and including a balanced ratio of and insecure codes.

## 3.7 Economic Analysis

In a commercial setting, the expense would be much more toward the engineering salaries and the expenses of hardware and API. There also would be maintenance costs as models would require re-training on new vulnerabilities, ongoing evaluation computational costs. As this study was conducted on personal hardware, there were no documented usage of energy consumptions by the system. However, the API cost from Google was around worth 900$ in credit. An LLM that can detect vulnerabilities like multiple SAST tools can save a lot of time. Not only that, it could also save expenses as some tools require licenses and it can be expensive. This can also help developers to save time for security audits and speed up deployment or productivity.

# Chapter 4

# Methodology and Design Specification

This chapter presents the framework used to analyze the security stance of AI-generated code in this chapter. It provides a thorough description of the research process from the motivations behind the specific dataset selected to the actual analysis protocols used, and finally to training the LLMs for mimicking the SAST tools. Each phase of the experimental process, with the tools, measures and processes employed for the analysis in both C and Python, are explicit in manner so as to provide transparency and ease of reproducibility. This chapter concludes with the formal specification of proposed framework for AI code security detection, the primary contribution of this work, incorporating the conceptual findings in a structured, developer-oriented approach.

## 4.1 Research Methodology and Experimental Design

This section describes the systematic process of data acquisition, preparation, and analysis that forms the foundation of this study's empirical work. It outlines the specific phases of the experiment, the tools employed, and the metrics used to evaluate the security of AI-generated code. Below is a methodology overview that goes through everything that has been done in this study. Reproducing the results of this study could be possible if the order and methodologies are as followed. Reproducibility of a study also contributes to a study. Thus, the overview and the details can be of great assistance.

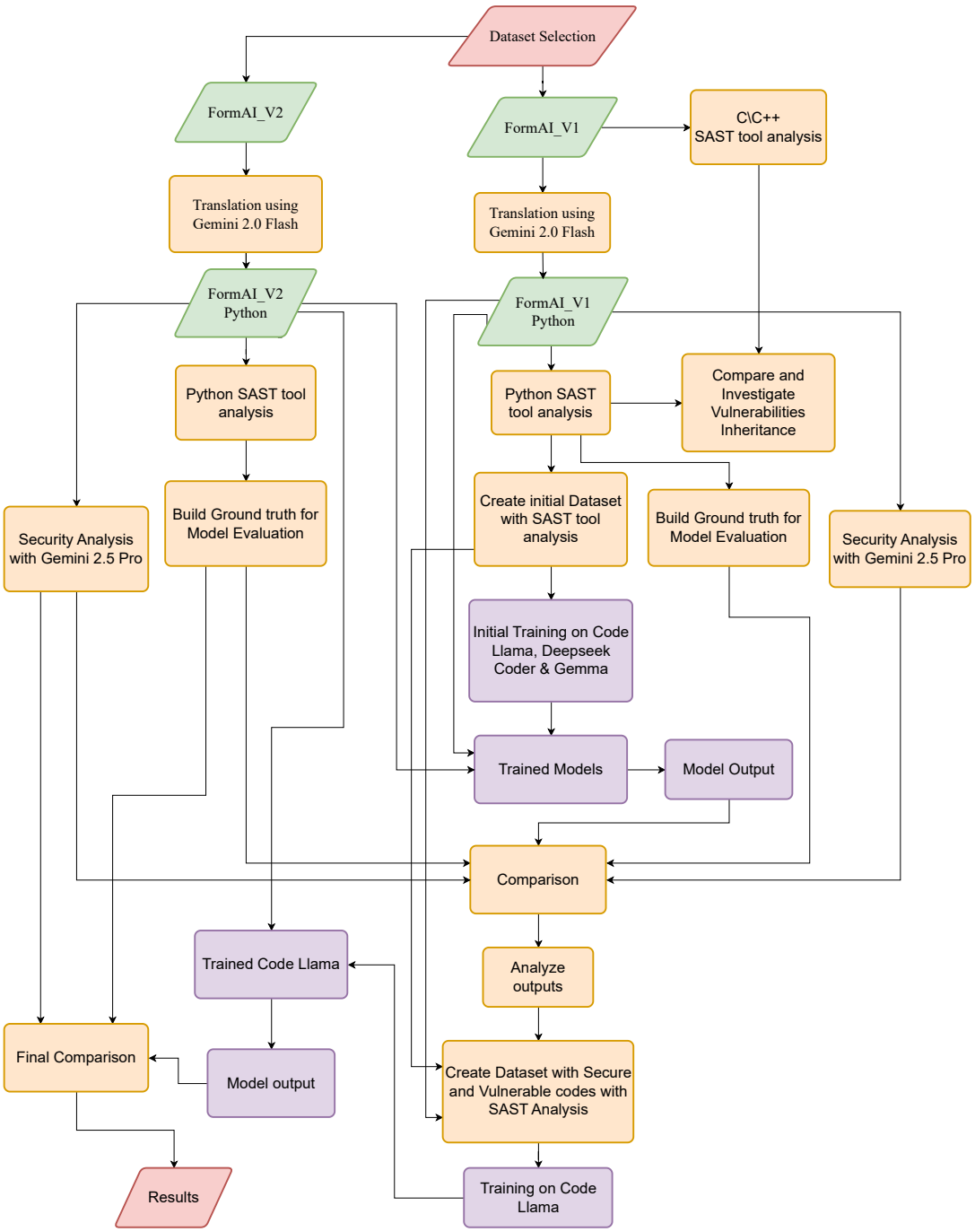



Figure 4.1: Methodology Overview

### 4.1.1 Dataset Selection and Justification

The initial research proposal outlined a plan to generate code samples from a variety of Large Language Models (LLMs). However, the methodology was refined due to constraints, such as computational resources and API access costs associated with generating and reliably labeling a new dataset of sufficient scale.

To keep the proposed study feasible and strong enough, it was decided to consider

the FormAI dataset [10] v1 was selected as the primary source of data. The FormAI v1 dataset is a collection of 112,000 AI-generated compilable and independent C programs created by the GPT-3.5-turbo model. This specific version was chosen over later releases to maintain methodological consistency. FormAI v1 contains code generated from a single LLM.

The vulnerability labels of the FormAI v1 dataset were produced using the Efficient SMT-based Bounded Model Checker (ESBMC). This approach identifies vulnerabilities by generating a "counterexample" for each detected flaw, a process which eliminates the possibility of false positives. For this research, the high-fidelity labels provided by ESBMC serve as the "ground truth" against which the performance of other static analysis tools is measured. The use of this dataset will allow this study for a comparative analysis of security tools.

### 4.1.2 Analysis Approach: Justification for Static Analysis

The initial research proposed the use of both Static (SAST) and Dynamic (DAST) Application Security Testing tools. However, the methodology was refined upon a detailed examination of the chosen datasets structure. Static Application Security Testing (SAST) analyzes an application's source code or binaries without executing the program. On the other hand, Dynamic Application Security Testing (DAST) observes a program's behavior and vulnerabilities while it is running.

The FormAI dataset v1 consists of 112,000 independent, compilable C programs, each designed to perform a self-contained task rather than function as part of a larger application. These programs lack the typical networked interfaces, such as API endpoints or web services, that DAST scanners require to interact with and test a running application. Given this fundamental incompatibility, DAST was deemed unsuitable for this study. Therefore, to ensure a valid and feasible analysis of the code as generated, the methodology focused exclusively on SAST, which is the appropriate approach for analyzing source code directly.

### 4.1.3 Static Analysis of Original C Code

This part of the study concentrated on performing a comparative static analysis of the original C code of the FormAI dataset v1. The goal was to compare how current mainstream SAST tools stacked up against the formal verification-based ground truth in the dataset provided by the ESBMC labels.

#### Tool Selection

In the static analysis of the C code, two open source tools are chosen which are Cppcheck and Flawfinder. Cppcheck is a static analysis tool that aims to identify a variety of programming errors, including most of those with direct security impact, like memory leaks and buffer overflows. Flawfinder is a security tool that does a simplistic scan of the code looking for functions or coding patterns known to be

associated with a variety of vulnerabilities [2]. Both a general bug-finder and this specialized security scanner were used for the purpose of analysis. A combined result would offer the best possible insight into what could otherwise simply appear as routine behavior from each individual tool [1] [5].

### Analysis Protocol

Scripts were used to execute the selected static analysis tools on each of the 112,000 C programs. The analysis protocol was executed as follows:

1. **Flawfinder Execution:** Each C program was analyzed by Flawfinder to perform a targeted search for security-specific weaknesses. The tool was executed with specific configurations and its output was directed to a CSV file for structured logging.

    - `$env:PYTHONIOENCODING="ISO-8859-1"; $env:PYTHONUTF8="0";`
      `$env:LC_ALL="C.ISO-8859-1";`
      `flawfinder --csv sample/ > result.csv`

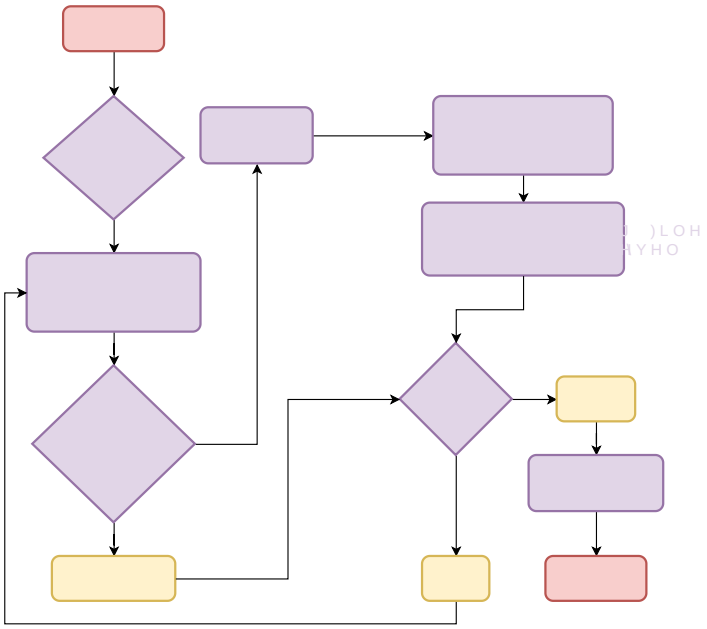


Figure 4.2: Flawfinder Analysis Workflow

2. **Cppcheck Execution:** Cppcheck identify a broader range of programming bugs and potential flaws. Cppcheck was configured to check against the C99 standard and generate a structured report in XML format.

    - `cppcheck --enable=all --inconclusive --std=c99`
      `--xml --xml-version=2 sample/ 2> sample.xml`

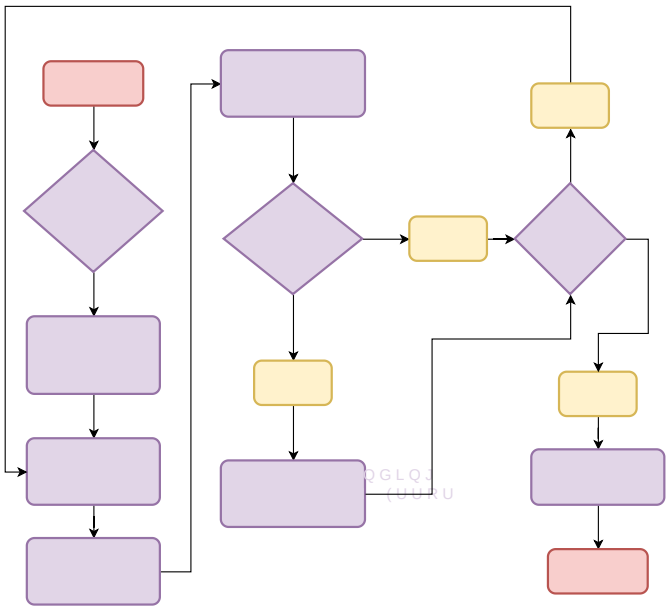


Figure 4.3: Cppchecker Analysis Workflow

The results of both Cppcheck and Flawfinder were normalized and compared to each corresponding file from the FormAI v1 dataset and prepare the data for comparative analysis.

#### Quantitative Performance Evaluation

The results from both Cppcheck and Flawfinder were compiled. The initial research plan involved performing a direct quantitative comparison of these results against the ESBMC ground truth labels using standard classification metrics such as Precision, Recall, and F1-Score.

However, the output formats and vulnerability descriptions differ between the tools and the ESBMC ground truth. A valid comparison would require normalization and validation to accurately correlate the findings.

### 4.1.4 Cross-Language Conversion and Analysis

This second phase of the experiment was to investigate the behavior of AI-generated code after it is translated to a different language. The objective was to determine how vulnerabilities manifest or change when code is converted from a low-level, memory-managed language (C) to a high-level, memory-safe language (Python).

#### C-to-Python Code Conversion

The entire collection of 112,000 C programs from the FormAI dataset v1 was converted to Python. The translation was performed using a Python script that

used the official Google Generative AI SDK to interface with the Gemini 2.0 Flash Large Language Model.

The script iterated through each C source file and submitted its content to the Gemini API. To guide the translation, the following carefully engineered prompt was used for each submission, with the {c_code_content} placeholder being dynamically replaced by the contents of each C file:

Translate the following C code into Python. The Python code should be functionally equivalent to the C code. Ensure the Python code is well-structured and follows Pythonic conventions (e.g., PEP 8). Do not include any conversational preamble or explanations in your response, only the Python code itself.If the C code includes comments, try to preserve them or create equivalent Python comments. C Code: {c_code_content} Python Code:

This prompt was designed for a direct code-for-code translation, and the instruction to omit conversational text was important for the successful automated parsing of the model's output.

The gap between C and Python is substantial, particularly concerning concepts like pointers, manual memory management, and explicit data typing in C versus Python's object model, garbage collection, and dynamic typing. It is recognized that a direct, one-to-one translation of every feature is not always possible, and the LLM's conversion may introduce new logic or omit original logic to create functional Python code. This process therefore simulates a real-world scenario where a developer uses use an AI assistant for development.

### Tool Selection for Python Analysis

For the static analysis of the translated Python code, five SAST tools was selected: Pylint, Bandit, Semgrep, Snyk, and Vulnhuntr. This multi-tool approach was designed to have a diverse range of analysis techniques, providing a more comprehensive assessment than any single tool could offer [7] [42] [16] [17] [18].

The selected tools and their specific roles are detailed below, categorized by their primary analysis technology:

- **Traditional Static Analysis Tools:** This group represents established SAST methodologies.
  - **Pylint (Linter-Based Analysis):** Pylint primarily enforces coding standards and identifies programming errors.

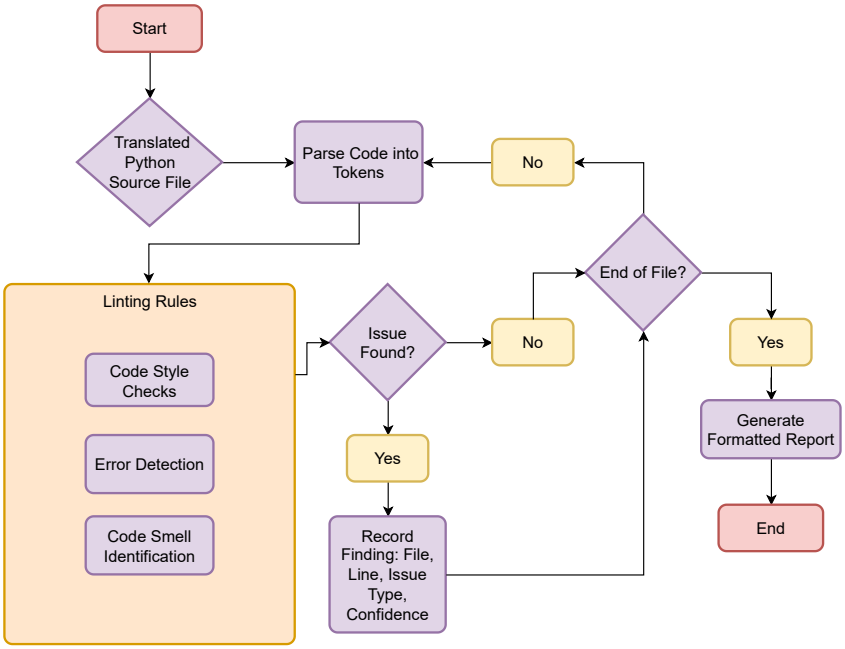



Figure 4.4: Pylint Analysis Workflow

– **Bandit (AST-Based Analysis):** An open-source security scanner, Bandit operates by parsing source code into an Abstract Syntax Tree (AST) and applying a predefined set of rules to the tree's nodes to find common Python security vulnerabilities.

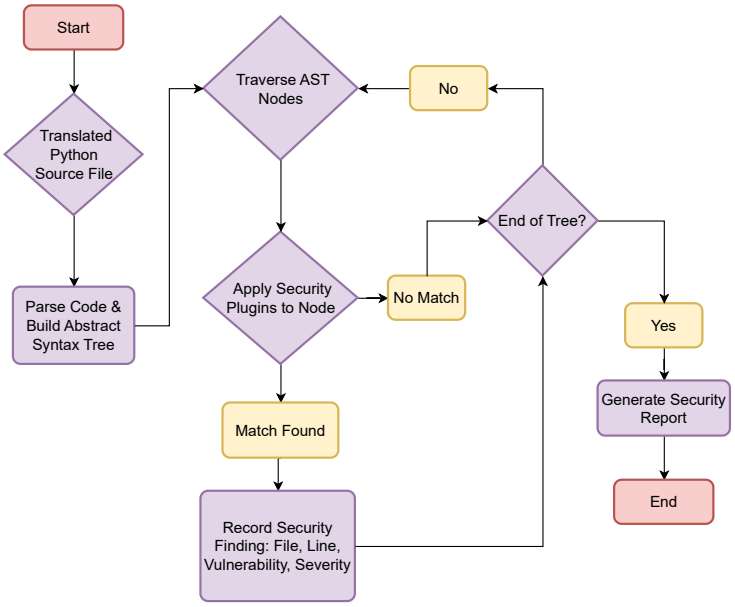



Figure 4.5: Bandit Analysis Workflow

– **Semgrep (Advanced Pattern Matching):** Included for its powerful, language-aware pattern-matching engine. Semgrep analyzes to detect

complex and specific vulnerability patterns defined by its extensive rule set, offering more flexibility than traditional scanners.

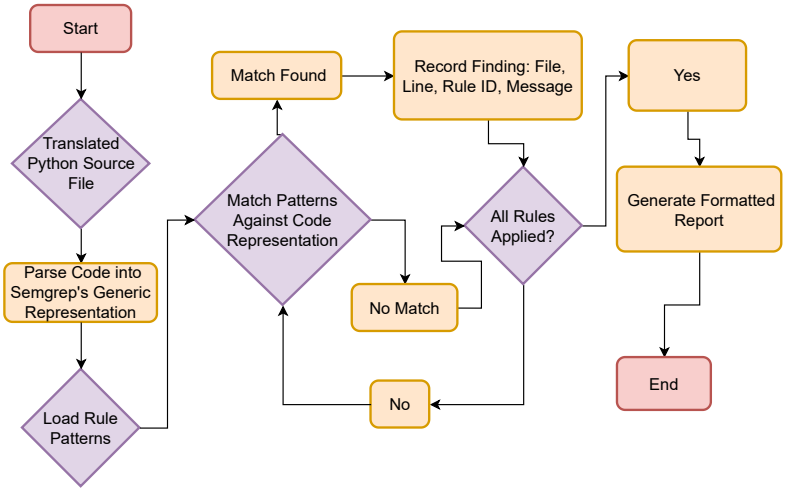



Figure 4.6: Semgrep Analysis Workflow

- **Hybrid and Advanced Analysis Tools:** This group represents modern, multi-faceted approaches to static analysis.
  - **Snyk (Commercial-Grade Hybrid Analysis):** Employed as an industry benchmark, Snyk's SAST engine uses a hybrid approach, combining multiple advanced techniques such as data-flow analysis with sophisticated pattern matching to provide high-fidelity vulnerability detection.

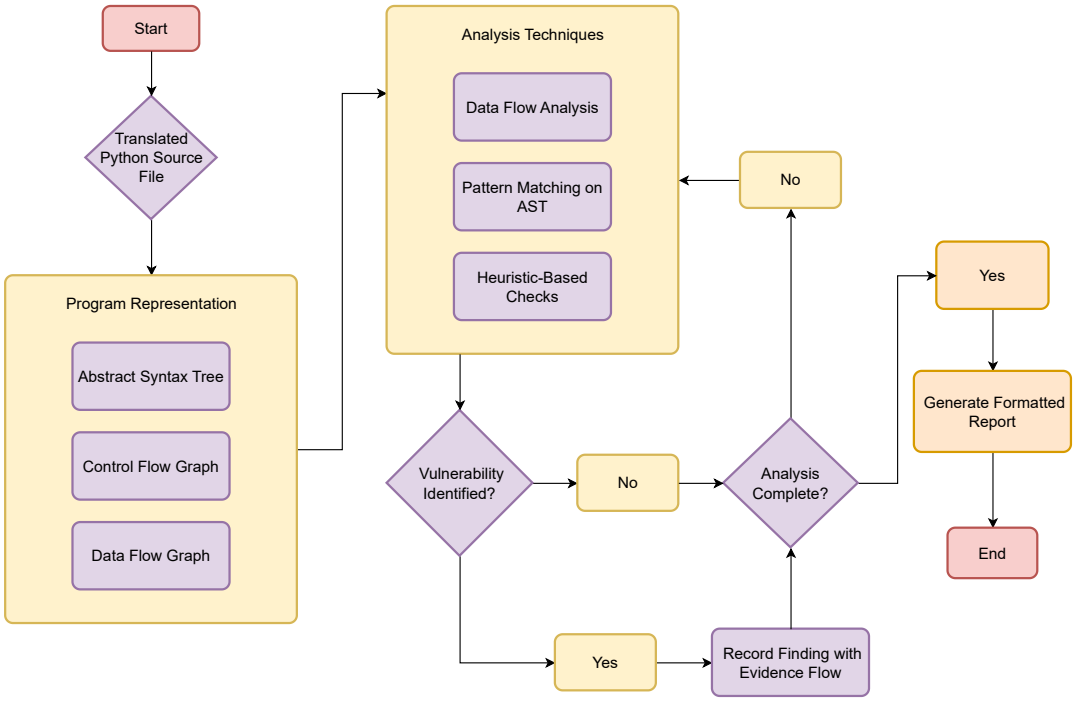



Figure 4.7: Snyk Analysis Workflow

– **Vulnhuntr (LLM-Augmented Hybrid Analysis):** Vulnhuntr was selected because of its hybrid SAST methodology. There is a two-staged process: first, the tool performed a rapid, local, rule-based scan to generate an initial list of potential vulnerabilities. Subsequently, each of these initial findings was then sent to the Gemini 2.0 Flash model via API. The LLM was tasked with analyzing the specific code snippet and vulnerability context to help verify the finding, thereby aiming to reduce false positives and provide deeper analytical insight.

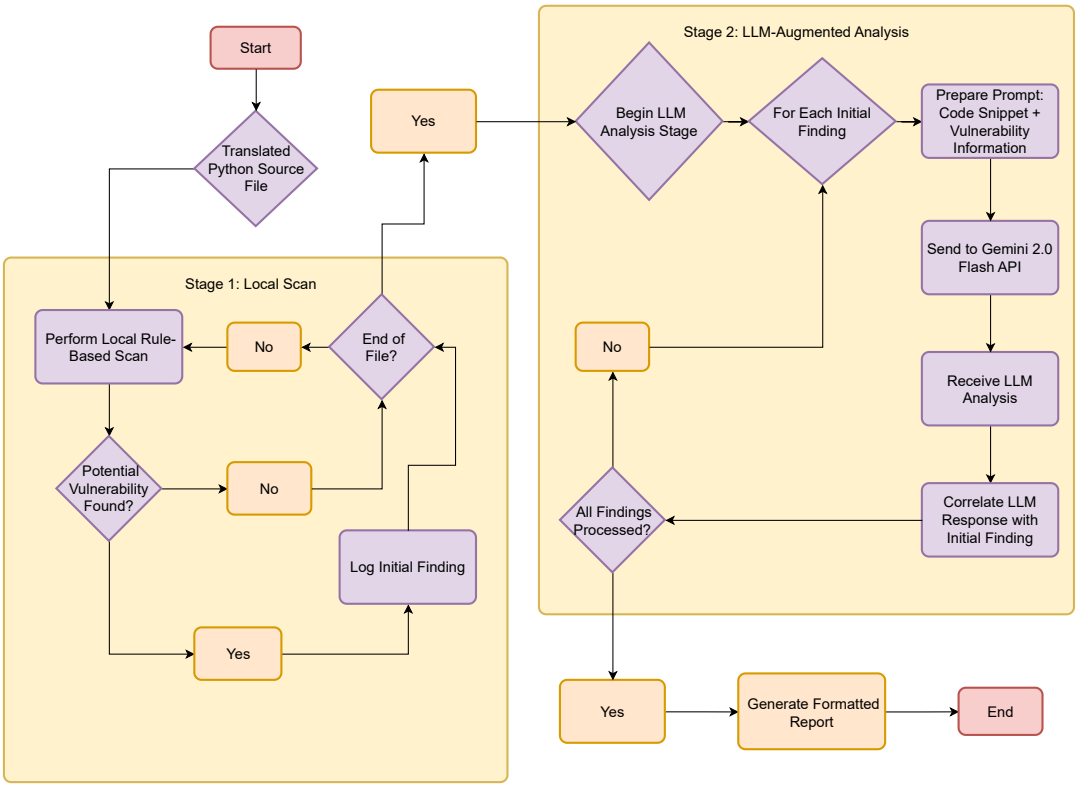



Figure 4.8: Vulnhuntr Hybrid Analysis Workflow

By employing this combination of tools, each with a different analytical approach, the research aimed to create a holistic view of the security landscape of the translated code, capturing a wider array of potential vulnerabilities through their distinct and complementary strengths.

### Analysis Protocol

The execution of each tool was handled sequentially to ensure that the findings could be logged correctly for each analysis type. The commands were executed as follows:

1. **Pylint Execution:** Each Python file was processed by Pylint to assess code quality and identify programming errors, with the results exported in a structured JSON format. The command used was:

```
pylint --output-format=json dataset > pylint_report.json
```

2. **Bandit Execution:** The codebase was recursively scanned by Bandit to find common security vulnerabilities, generating a report in CSV format. The command used was:

   ```
   bandit -r -f csv -o bandit_report.csv dataset/
   ```

3. **Semgrep Execution:** Semgrep was run against the Python files using a curated, community-driven ruleset for Python (p/python) to identify a wide range of vulnerabilities, with the output saved as a JSON file. The command was:

   ```
   semgrep --config "p/python" --json -o
   semgrep_report.json /dataset
   ```

4. **Snyk Execution:** A scan was initiated using the Snyk command-line interface (CLI) to analyze the code with its proprietary SAST engine. The command was executed in the root directory of the codebase:

   ```
   snyk code test
   ```

5. **Vulnhuntr Execution:** Finally, the hybrid analysis was performed by Vulnhuntr. The tool was configured to use its local scanner and augment the analysis by sending findings to the Gemini LLM for verification, with the final report saved to a text file. The command used was:

   ```
   vulnhuntr --root dataset --llm gemini > vulnhuntr_report.txt
   ```

The results from these five tools outputs were systematically recorded to compile an overall, multi-faceted list of possible vulnerabilities present within the translated Python code ready for collection and analysis.

### Results Collation

The outputs from the five distinct Python analysis tools were generated in various formats, including JSON, CSV, and plain text. As a result, a data normalization process was required. Custom Python scripts were developed to read report formats from each tool and log the findings into a single database.

For each translated Python file, this database was structured to log the vulnerabilities reported by each of the five tools. This approach allowed for a cross-tool comparison, making it possible to identify the difference between the tools. The process focused on documenting which vulnerabilities were detected by multiple tools versus those identified by only a single scanner. The final dataset serves as the basis for the results and analysis presented in the subsequent chapter.

### 4.1.5 Software and Tools Environment

All experiments were conducted within a consistent software environment to ensure the reproducibility of the findings. The primary development and scripting language used was Python, with version 3.12 used for the main analysis scripts and version 3.10 used specifically as required for the Vulnhuntr tool.

The code translation phase was facilitated by the Google Generative AI SDK (version 0.8.5), which provided the interface to the Gemini 2.0 Flash model.

For the static analysis portions of the study, the latest stable versions of the following tools available during the experimental period (circa Q2 2025) were utilized:

- Cppcheck
- Flawfinder
- Pylint
- Bandit
- Semgrep
- Snyk Command Line Interface (CLI)
- Vulnhuntr

The Vulnhuntr tool used was a recent version that was personally modified by the author to replace its default LLM API integration with a custom implementation that specifically calls the Google Gemini 2.0 Flash model, ensuring consistency with the code translation phase.

## 4.2 Design Specification: A Methodology for LLM Fine-Tuning

This section presents the design specification for a methodology aimed at identifying the security vulnerabilities of code-generating Large Language Models (LLMs). The core of this methodology is the use of a multi-layered dataset, created during the experimental work of this thesis, to fine-tune base LLMs to produce an analysis inherently identical to that of the SAST tools. This specification serves as the formal proposal and blueprint for the next phase of this research .

### 4.2.1 Design Rationale

The work detailed in this thesis and the broader literature confirms that LLMs such as the GPT-3.5-turbo model that generated the FormAI v1 dataset, frequently produce code with significant security vulnerabilities. While analyzing this output is necessary, a more fundamental solution is to address the root cause by improving the code generation process itself.

The experiment served not only to evaluate existing SAST tools but also to create a unique dataset. This new dataset, containing the AI-translated Python translation, and a multi-tool security analysis of the Python code, is a unique resource. The proposed methodologies are therefore designed to use this asset to fine-tune existing, capable, open-source LLMs to be more security-aware.

### 4.2.2 The New Python Security Dataset

One of the contributions of the experimental phase is a new dataset derived from the FormAI v1 corpus. This dataset, which serves as the foundation for the proposed fine-tuning methodologies, is composed of the following steps for each of the 112,000 samples:

1. Original C Code: The source C code as generated by GPT-3.5-turbo.
2. Formal Verification Ground Truth: The corresponding ESBMC vulnerability labels for the C code, providing a high-fidelity record of its security flaws.
3. Translated Python Code: The Python equivalent of the C code, as translated by the Gemini 2.0 Flash model.
4. Multi-Tool SAST Analysis: The collated and normalized vulnerability reports for the Python code from five distinct SAST tools (Pylint, Bandit, Semgrep, Snyk, and Vulnhuntr), providing a multi-perspective security assessment.

### 4.2.3 Proposed Fine-Tuning Methodology

The enriched dataset enables the application Supervised Fine-Tuning (SFT) for Vulnerability detection.

#### Supervised Fine-Tuning (SFT) for Vulnerability Remediation

This methodology aims to directly teach the LLM how to correct vulnerable code. The process involves creating a high-quality dataset of (vulnerable_code, secure_code) pairs [22].

- Data Preparation: For each Python file identified as vulnerable by the multi-tool analysis in Layer 4, the findings will be categorized and used in the training dataset in order to mimic the SAST tools analysis.
- Training: The LLM will be trained on this dataset of paired examples. The model learns the patterns that the tools use to identify the vulnerabilities. Thus, learning to identify vulnerabilities just like the SAST tools.
- Goal: The resulting model is expected to be able to identify security vulnerabilities just like how the SAST tools would detect vulnerabilities.

### 4.2.4 Model Selection

Three models that were both open source and trusted by community were used in this study. The models were trained and fine tuned to identify the vulnerabilities like the SAST tools. The 7B and 6.7B parameters for the models were used because of the computational resources that are available. Higher parameters would require more resources that are currently unavailable.

- Code Llama 7B Instruct
- Deepseek Coder 6.7B Instruct
- Gemma 7B Instruct

Additionally, the Gemini 2.5 pro model was also utilized by using its API. Prompt engineering was used only in this case as training a commercial model is not possible.

### 4.2.5 Dataset Development and Structuring

A supervised fine-tuning (SFT) approach was utilized, which required the creation of structured training dataset. The process of dataset creation was iterative, due to the evolution in strategy based on initial results.

**Dataset 1: SAST-Informed Vulnerability Snippet Dataset**

The initial training dataset was constructed directly from the outputs of the SAST tools used in the analysis phase. Because due to VRAM limitation both secure codes and vulnerable codes could not be incorporated into the training dataset even to the lowest. Thus adopting to SAST generated results only in order to minimize token inputs. Each entry in this dataset followed a three-part structure:

- **Instruction:** A prompt tasking the model to identify the vulnerability in the provided code.
- **Input:** The specific lines of Python code flagged by a SAST tool.
- **Output:** The corresponding vulnerability type reported by the tool.

All three models (Llama, Deepseek, Gemma) were initially trained on this dataset.

**Dataset 2: Secure vs. Vulnerable Code Classification Dataset**

Preliminary evaluations of models trained on the Phase 1 dataset revealed lower-than-expected precision. Therefore, in order to increase precision, adding secure code was necessary. This new dataset consisted of 7,778 full Python code samples. The number is lower as input tokens were thoroughly optimized while it was being reduced. It was balanced, with approximately half of the samples containing known vulnerabilities and the other half deemed secure. This dataset also followed a three-part structure:

- **Instruction:** A prompt tasking the model to identify the vulnerability in the provided code.

- **Code:** The entirety of the Python code snippet.
- **Response:** The response would be either secure, or the category of vulnerabilities that were identified by the model.

### 4.2.6 Supervised Fine-Tuning Process

The fine-tuning process was conducted iteratively:

- **Initial Training:** All three selected models (Llama, Deepseek, Gemma) were fine-tuned on the Training Dataset 1.
- **Hyperparameter Tuning:** Based on initial performance, the training arguments and hyperparameters were adjusted, and the models were retrained on the same Training Dataset 1 to optimize performance.
- **Final Training on Refined Dataset:** After evaluating the retrained models, the Llama Instruct model was identified as the best performer. Consequently, only this model was subjected to a final round of fine-tuning using the higher-quality Training Dataset 2 to enhance its security classification capabilities.

# Chapter 5

# Implementation

This chapter explains the practical and technical aspects and working pipeline of the experiments. Here it is explained how the methodologies are implemented and how the results would be produced. The goal is to ensure the transparency and reproducibility of the research.

## 5.1 System Environment and Tooling

The fine-tuning and evaluation processes were conducted on a system with the following specifications:

- Hardware:
  - GPU: Nvidia RTX 3080 10GB
  - CPU: AMD Ryzen 7700
  - RAM: 32GB
- Software:
  - Operating System: Microsoft Windows 11
  - Core Framework: PyTorch
  - Hugging Face Ecosystem:
    - Transformers: For model loading and tokenization.
    - PEFT: For implementing the Parameter-Efficient Fine-Tuning (PEFT) techniques.
    - bitsandbytes: For 4-bit quantization (QLoRA).
  - SAST Tools: Bandit, Semgrep, Snyk and Vulnhuntr
  - IDE: Visual Studio code & Cursor
  - Python version 3.10.11

## 5.2 Model Configuration

Three models were used in this research: Code Llama 7B Instruct, Gemma 7B Instruct and Deepseek Coder 6.7B Instruct. In order to run these model, QLoRA (Quantized Low-Rank Adaptation) technique was applied. What this does:

- Loading the Base Model: The pre-trained base models (Llama, Deepseek, Gemma) were loaded in a 4-bit quantized format. This significantly reduces the memory usage.
- Attaching LoRA Adapters: Small, trainable "adapter" layers were attached to specific modules of the frozen, quantized base model.
- Training: During the fine-tuning process, only the parameters of these small adapter layers were updated. The vast majority of the model's original parameters remained frozen.

This approach allows for models to be loaded with less than 24GB of VRAM per model, without significantly degrading the base model's performance too much.

Below are some sample code snippets:

```
tokenizer = AutoTokenizer.from_pretrained(
    BASE_MODEL_NAME,
    trust_remote_code=True
)
tokenizer.pad_token = tokenizer.eos_token
quant_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_use_double_quant=False
)
```

## 5.3 Preparing Dataset 1 with SAST results

At first, the intention was to train the models with FormAI V1 Python code files and their corresponding SAST results. There were around 45,000 secure codes. therefore, to balance the dataset, 45,000 secure and 45,000 vulnerable codes were taken. In total there were around 90,000 python codes loaded with their SAST security analysis. However, this proved to be a challenge because this huge amount of dataset was not loading into the memory of the GPU. The VRAM quickly filled up and crashed the system. To fix this, the amount of codes that were being loaded into the dataset was reduced. The amount was reduced by 5,000 each time and even though it was down to 10,000 codes (5,000 secure and 5,000 vulnerable) it could not get the model to start training. Sometimes it did start but soon the VRAM filled up and crashed the system.

Therefore, in order to train the models to analyze code snippets like the SAST tools, it was decided to train the models using the SAST tools generated results. In this method, the training dataset was loaded with only the SAST tools verdicts and their corresponding code snippet. Specifically, the training dataset had the information as to which part of the code was vulnerable and what kind of vulnerability it was.

Excluding Pylint, which had significant detections of over 800,000 that could bias the model, there were around 40,000 results on which the model would train. This

training dataset was designed to perfectly mimic how the SAST tools operate and detect vulnerabilities.

## 5.4 Training Models on Dataset 1

To start the training, the trainer and training arguments need to be set first at least to their default state in order to start training the model. But first some test prompts were run to at least make sure it was working. After setting trainer and training arguments the training dataset is loaded. The dataset was loaded into the model perfectly and during training the VRAM usage was well around 12.67GB. Each model had an average training time of 90 minutes. Below are sample snippets of first training codes:

```
training_args = TrainingArguments(
    output_dir=NEW_MODEL_NAME,
    num_train_epochs=1,
    per_device_train_batch_size=1,
    gradient_accumulation_steps=1,
    save_steps=5000,
    logging_steps=25,
    learning_rate=2e-4,
    fp16=True,
    max_grad_norm=0.3,
    max_steps=-1,
    warmup_ratio=0.03,
    lr_scheduler_type="constant",
)
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    dataset_text_field="text",
    max_seq_length=512,
    tokenizer=tokenizer,
    args=training_args,
    formatting_func=lambda example: f"<s>[INST]
    {example['instruction']} \n\n{example['input']}
    [/INST] {example['output']} </s>"
)
```

## 5.5 Evaluating base Model on Random 5000 Sample

After the training, the evaluation was run twice. Once on the FormAI V1 Python codes and next on the V2 codes. Initially, the intention was to run the models on all of the codes. But the estimated time to finish it showed was over 800 hours. It was not possible to conduct a thorough evaluation so compromises were made and random 5,000 samples were chosen out of both V1 and V2 code samples. Gemini 2.5

Pro API was also used to evaluate in order to demonstrate how a generalized public model would work.

In order to evaluate, trained adapter was used on the base model so that it works as it was trained. Next, the model was loaded with instructions and random 5,000 samples to generate output. Sample code that does this work is given below:

```
for filepath in tqdm(files_to_process, desc="Analyzing files"):
    with open(filepath, 'r', errors='ignore') as code_file:
        content = code_file.read()
    if not content.strip(): continue
    prompt = f"<s>[INST] {instruction} \n\n{content} [/INST]"
    input_ids = tokenizer(prompt, return_tensors="pt",truncation=True,
    max_length=4096).input_ids.cuda()
    outputs = model.generate(input_ids=input_ids, max_new_tokens=256)
    result = tokenizer.decode(outputs[0], skip_special_tokens=True)
    analysis = result.split("[/INST]")[1].strip()
    relative_path = os.path.relpath(filepath, codebase_path)
    writer.writerow([relative_path, analysis])
```

After the evaluation is complete a CSV file is generated that needs to be processed for result comparison later on.

## 5.6 Optimized Training Models on Dataset 1

The first result generated was indeed not satisfactory. Therefore, optimizations were needed. To optimize the trainer and the training argument, hyperparameters were changed so that the model improves. Sample of the trainer and training arguments code snippet is given below:

```
training_args = TrainingArguments(
    output_dir=NEW_MODEL_NAME,
    num_train_epochs=3,
    per_device_train_batch_size=1,
    gradient_accumulation_steps=4,
    evaluation_strategy="steps",
    eval_steps=200,
    save_strategy="steps",
    save_steps=200,
    load_best_model_at_end=True,
    save_total_limit=2,
    logging_steps=25,
    learning_rate=2e-4,
    fp16=True,
    max_grad_norm=0.3,
    max_steps=-1,
    warmup_ratio=0.03,
    lr_scheduler_type="constant"
)
```

```
trainer = SFTTrainer(
    model=model,
    args=training_args,
    train_dataset=split_dataset["train"],
    eval_dataset=split_dataset["test"],
    peft_config=peft_config,
    tokenizer=tokenizer,
    max_seq_length=1024,
    formatting_func=lambda example: [f"<s>[INST]
    {example['instruction']} \n\n{example['input']}
    [/INST] {example['output']} </s>"]
)
```

Epoch, increased gradient accumulation steps, evaluation steps, and some other hyperparameters were changed. This training took approximately 250 minutes to complete with a VRAM usage of around 18.8GB. To prevent overfitting the models, the hyperparameters were kept as optimal as possible through trial and error.

## 5.7 Evaluating Optimized Model on Random 5000 Sample

In the same manner as before, models are evaluated twice, both on V1 and V2 on randomly selected 5,000 samples. However, from the output of the previous models, prompt engineering was needed for Deepseek Coder, Gemma and Gemini API. Code Llama gave the cleanest output so it was needed to make sure other models do not incorporate extra texts such as recommendations, comments, the part of the codes that are vulnerable etc. Here are the prompts from the optimized versions:

**Llama:**

```
instruction = "Analyze the provided Python code and identify
security vulnerabilities."
```

**Deepseek Coder:**

```
instruction = """Analyze the following Python code for security
vulnerabilities.
- List the identified vulnerabilities using a bulleted list
(e.g., using '-').
- Be concise and do not include conversational text.
- If no vulnerabilities are found, please state 'No vulnerabilities
found.'"""
```

**Gemma:**

```
instruction = "You are a specialized AI assistant for software
security."
user_instruction = f"""Analyze the following Python code for
securityvulnerabilities.
```

```
- List the identified vulnerabilities using a bulleted list
(e.g., using '-').
- Be concise and do not include any other text.
- If no vulnerabilities are found, please state 'No
vulnerabilities found.' Code: ‘‘‘python {content} ‘‘‘ """
```

**Gemini API:**

```
prompt = f"""
You are an expert software security analyst. Your task is to
perform a security audit of the following Python code.
Analyze the code for any potential security vulnerabilities,
such as command injection, SQL injection, insecure cryptography,
path traversal, or buffer overflows.
If you find one or more vulnerabilities, describe each one clearly.
If you are confident that the code is secure and contains no
vulnerabilities, you must respond with the exact phrase:
"No vulnerabilities found."
Python Code: ‘‘‘python {content} ‘‘‘ """
```

The outputs were much more organized was easier to understand and categorize after some prompt engineering. Examples of the outputs will be shown later in Result and Analysis.

## 5.8 Preparing Dataset 2 with Secure and Vulnerable Codes

After optimization, the recall seemed to improve a lot. But the precision remained disappointing. Therefore, further training the models on secure codes was necessary.

To train on codes that use too many tokens on input, conducting a token analysis was required. The analysis results will be shown later on. The token analysis gave two recommended input token count. the 99th percentile and the 95th percentile. Previously, the model could not be loaded with 10,000 samples. So the token analysis was started with 10,000 code samples in the training dataset while Using the recommended values from 99th percentile to 95th percentile only if failed. This time it worked but as the training went on and the models saved the training progress, the VRAM would keep getting full much later in time and crashes. During 10,000 code training dataset the estimated time of completion was 51 hours. It took time for the VRAM to get full during saving steps so it crashes much later down the line. These parameters were to be adjusted slowly, reducing these by 1,111 samples and re-running the token analysis. However, each time It was crashing after many hours during saving as it occupied the VRAM and filling it up much faster. Here is the error code that was shown during those time:

```
File "G:\thesis\work\venv\lib\site-packages\transformers\models\llama
\modeling_llama.py", line 145, in forward
    sin = emb.sin()
torch.OutOfMemoryError: CUDA out of memory. Tried to allocate 2.00 MiB.
```

```
GPU 0 has a total capacity of 10.00 GiB of which 0 bytes is free. Of
the allocated memory 19.49 GiB is allocated by PyTorch, and 4.52 GiB
is reserved by PyTorch but unallocated.
```

Consequently, after reducing to 7,778 samples with an input of 95% tokens, the training successfully completed. The input token was capped at 1425 tokens and the training completed after around 33 hours and 38 minutes.

## 5.9 Training and Evaluating Fine-tuned Model

As Code Llama was the best performing and best instruction following model, it was decided to fine-tune llama only. During training the only change was on how the model would perceive the training dataset as it was changed. After training, to stay true to the proposed methodology, the evaluation was run on the FormAI V2 Python samples which were unseen to the model. Due to time limitation it was not possible to run it on the training dataset itself but as proven previously, it would perform better on the codes it is trained upon. Consequently, with slight change to Code Llama's prompt, evaluation was conducted. The prompt change is below:

```
prompt_template = (
    "### INSTRUCTION:\n"
    "Analyze the following Python code for security vulnerabilities. "
    "If the code is secure, respond with the single word 'secure'. "
    "If vulnerabilities are present, respond with a comma-separated
    list of "
    "all vulnerability categories you have identified.\n\n"
    "### CODE:\n"
    "{code}\n\n"
    "### RESPONSE:\n"
    )
```

Because of this a very concise CSV is generated which is both faster to generate and easier to read. However, this time Code Llama hallucinated and called almost every code secure but at the same time it also detected vulnerabilities on the same code as well. It happened because of the training dataset was not fed entirely into the model during training. To combat this a filter was used to address the hallucination which will be demonstrated later on.

# Chapter 6

# Results and Analysis

This chapter presents the empirical results derived from the two-phase static analysis experiment detailed in the preceding chapter. The findings are presented sequentially, commencing with a high-level overview of the vulnerabilities detected by the full suite of analysis tools. This is followed by a focused examination of the results from the C code analysis and a detailed, multi-perspective report on the findings from the Phase AI-translated Python code analysis. The data presented here informs the quantitative foundation for the discussion and conclusions of this research.

## 6.1 High-Level Overview of Findings

The first analysis gives a big picture concerning the total amount of issues and their dispersion among the eight static analysis tools. This high-level summary sets the stage for framing the scope of possible bugs (weaknesses) in the original AI-generated C code and its AI-transformed Python counterpart.

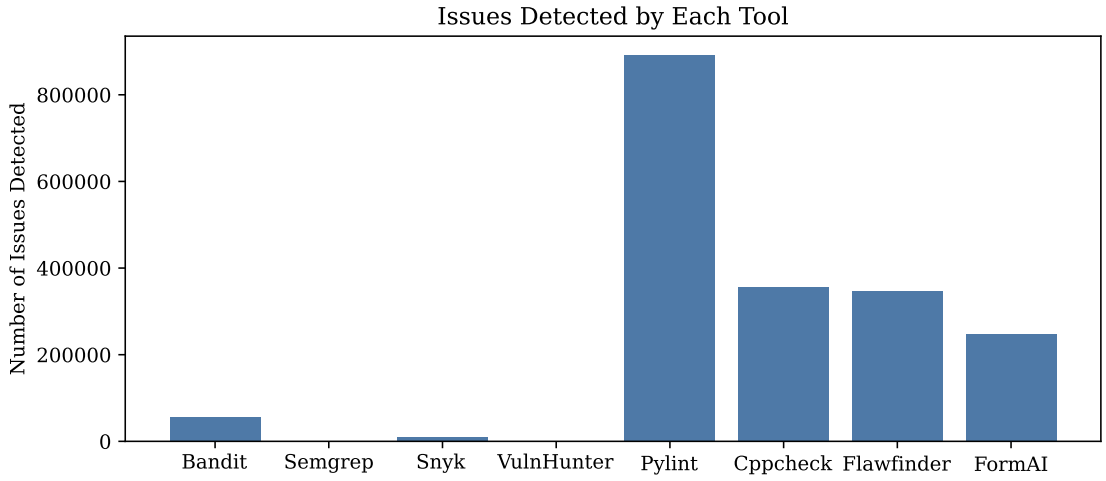



Figure 6.1: Issues Detected by Each Tool

A major difference is observed in the total number of issues each tool reports as seen in Figure 6.1. The linters and general-purpose bug-finders found the most issues –

Pylint reported 890,822 of them, Cppcheck 354,811, and Flawfinder 346,997. This is not surprising as these tools are made to catch anything wrong with code (quality, format, programming mistake) as well as security flaw. By comparison, the custom security scanner(s) – Semgrep (1,015) and Vulnhuntr (806), only come up with a much smaller and tighter set of findings.

A short comparison of total issue counts in the set of C and C++ tools and the Python tools sheds an interesting light. Figure 6.2 reveals that the total number of issues found in the AI-translated Python code (957,515) by the toolsets is in the same order of magnitude as that found in the original C code (948,357) by its toolset. This initial finding suggests that the process of automated code translation does not inherently reduce the overall volume of detectable issues, but rather shifts their nature and distribution, a subject that will be explored in subsequent sections.

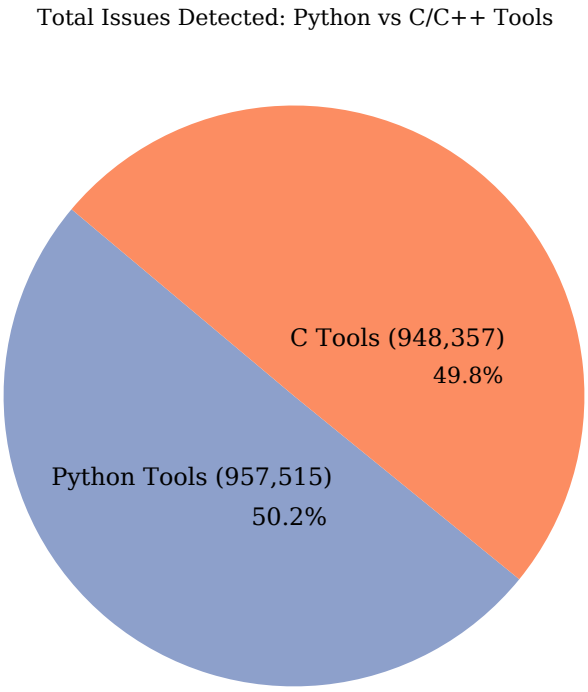



Figure 6.2: Total Issues Detected: Python vs C/C++ Tools

## 6.2 Analysis of Original C Code

The first experimental phase focused on analyzing the original 112,000 C programs from the FormAI v1 dataset. The SAST tools Cppcheck and Flawfinder were used, and their results were compared against the ESBMC-provided ground truth from FormAI.

**FormAI (Ground Truth):** Finally, an analysis of the ground truth data from the FormAI v1 dataset itself provides a baseline for the types of verifiable vulnerabilities

present in the original C code. The most common vulnerability class identified by the ESBMC formal verification tool was BUFFER_OVERFLOW_SCANF (86,472 instances), which is related to unsafe data input. This is followed by a series of dereference failures, indicating a high prevalence of pointer and memory management errors in the AI-generated C code.

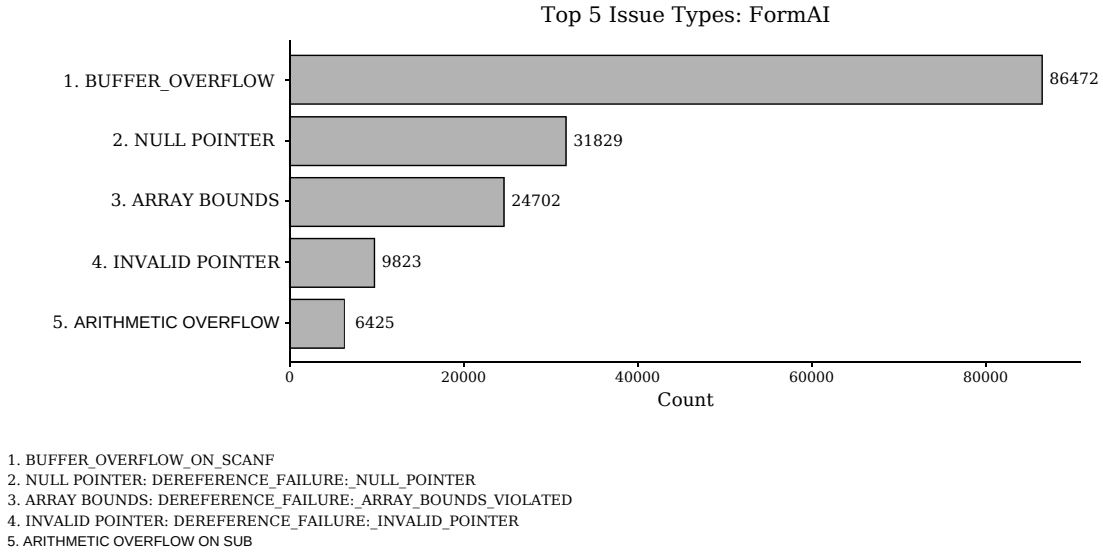



Figure 6.3: Top 5 issues: FormAI

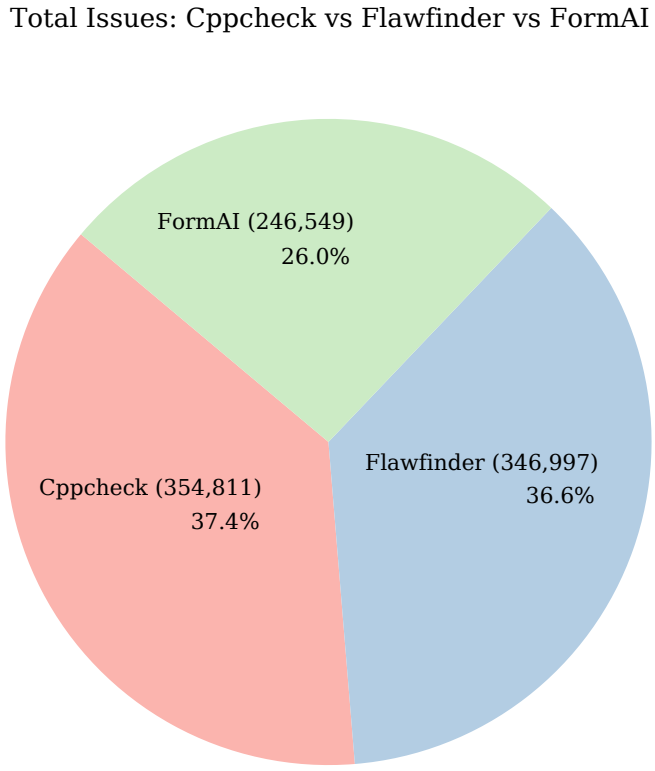
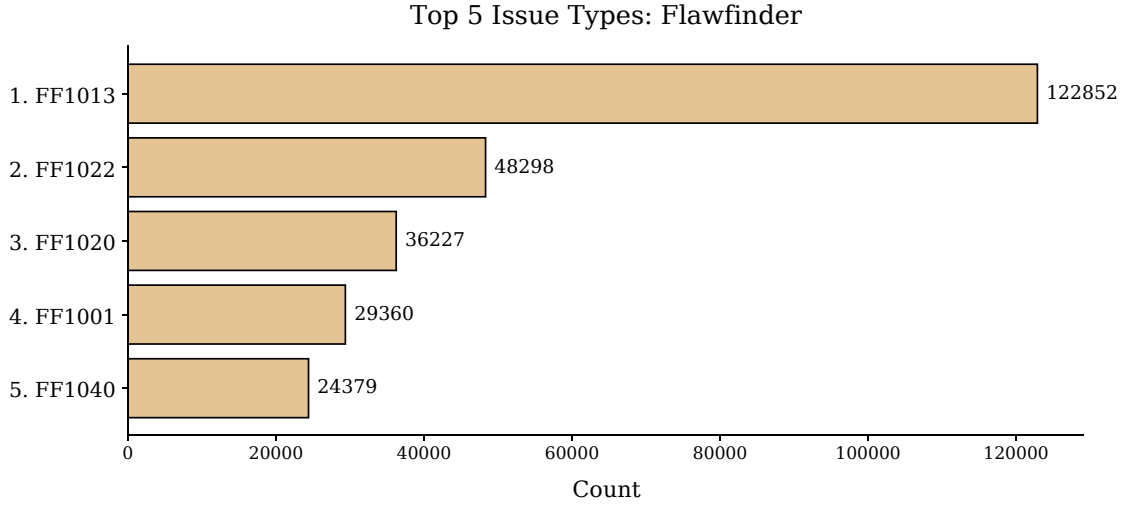



Figure 6.4: Total Issues Detected in C Code by Tool

As shown in Figure 6.4, both SAST tools identified a higher number of total issues (Cppcheck: 354,811; Flawfinder: 346,997) than the FormAI ground truth (246,549). This discrepancy is anticipated, as the ESBMC formal verification method is highly precise and targets specific, verifiable security properties, whereas SAST tools employ broader, pattern-matching rule sets that detect a wider range of potential bugs, style issues, and lower-severity warnings in addition to critical vulnerabilities.




1. FF1013: CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer
2. FF1022: CWE-126: Buffer Over-read
3. FF1020: CWE-20: Improper Input Validation
4. FF1001: CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
5. FF1040: CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization

Figure 6.5: Top 5 Most Frequent Issues Detected by Flawfinder

**Flawfinder:** A breakdown of the most frequent issues reported by each tool provides further insight. Flawfinder (Figure 6.5) primarily identified high-risk issues related to buffer overflows and unsafe string formatting. The top 5 issues code names are

- FF1013: CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer
- FF1022: CWE-126: Buffer Over-read
- FF1020: CWE-20: Improper Input Validation
- FF1001: CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
- FF1040: CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')

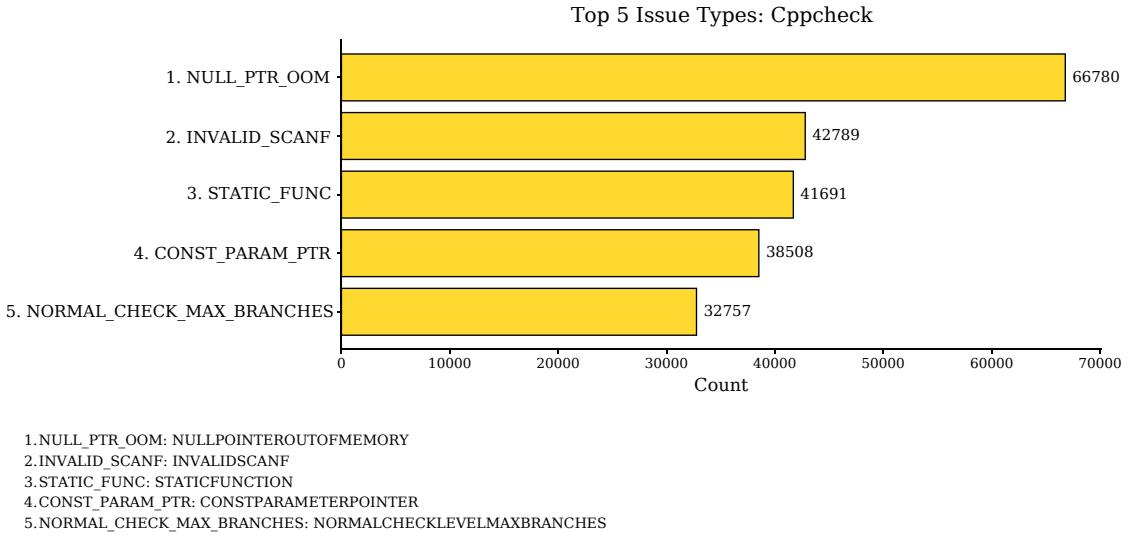



Figure 6.6: Top 5 Most Frequent Issues Detected by Cppcheck

**Cppchecker:** The analysis of Cppcheck's output (Figure 6.6) presented a different profile. The vast majority of its reports were categorized as NAN (353,195), which represents unclassified warnings or informational messages that could not be mapped to a standardized vulnerability type by the analysis script. The subsequent most frequent findings appear to be related to specific variable names from the dataset's programs rather than general vulnerability classes. This indicates that while Cppcheck is a powerful bug-finder, its output is less focused on standardized security weaknesses compared to a dedicated tool like Flawfinder, and requires significant normalization to be used for security-specific analysis.

## 6.3 Analysis of AI-Translated Python Code

The second phase of experimentation consisted in an examination of the translated Python code, which was generated from the original C source files. The study used a set of five SAST tools with various capabilities and domain focus to capture a wide variety of problems, including not only code style and good practice rule violations, but also severe security issues.

### 6.3.1 Findings of Each Tool Individually

An analysis of the output of the individualtool provides with the fine-grained view of the kind of issues found in the AI-translated Python code.

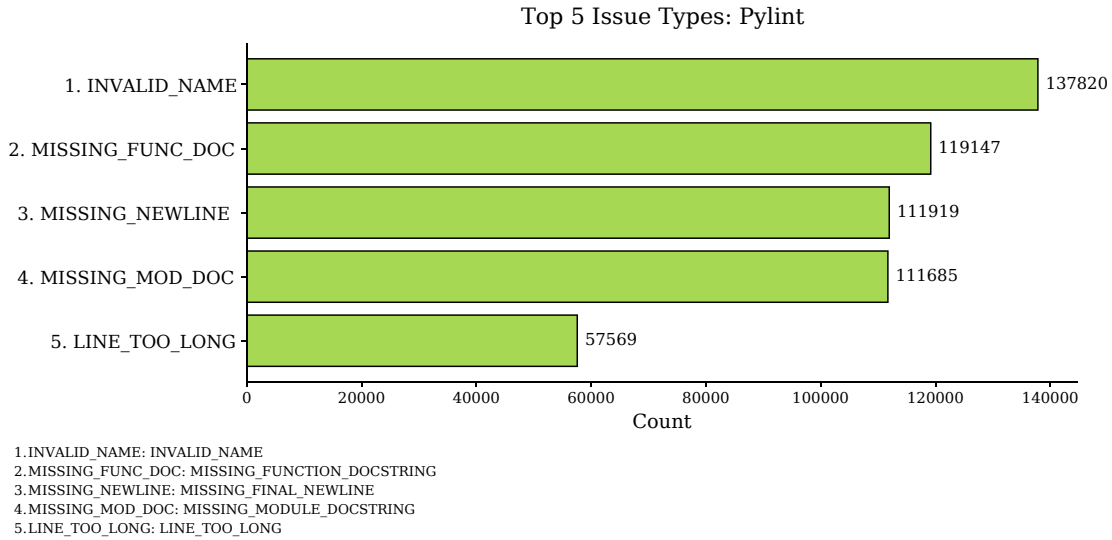



Figure 6.7: Top 5 Most Frequent Issues Detected by Pylint

**Pylint:** As a linter, Pylint's findings are heavily skewed towards code quality and convention issues. As seen in Figure 6.7, the second most common reported issue by a significant margin was MISSING_MODULE_DOCSTRING (111,269 instances), followed by other style and formatting warnings. This highlights that the LLM, while capable of translating functional code, does not inherently generate well-documented or PEP 8-compliant code without specific prompting.

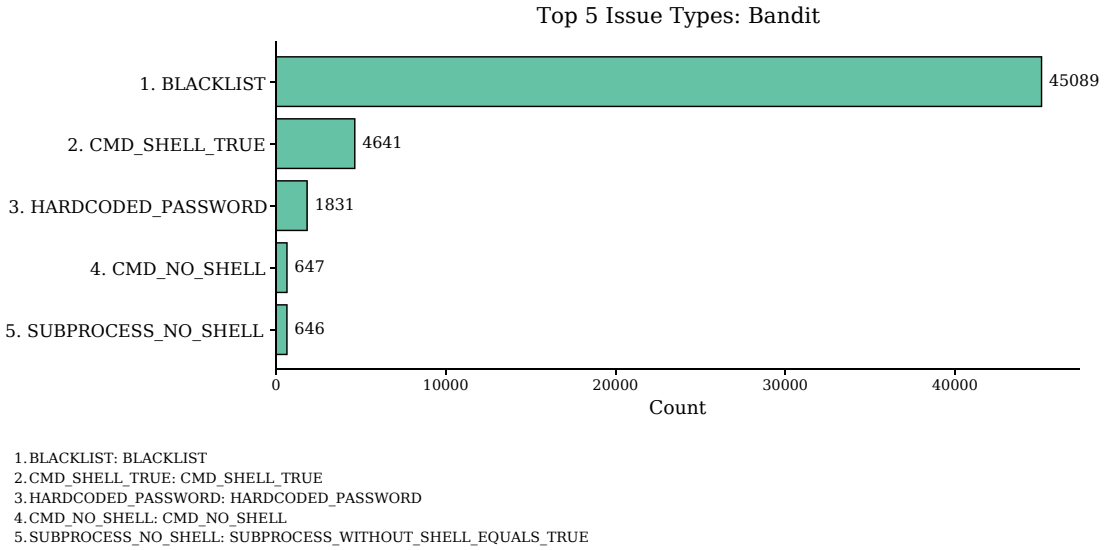



Figure 6.8: Top 5 Most Frequent Issues Detected by Bandit

**Bandit:** The AST-based security scanner Bandit focused on identifying common security anti-patterns. Its most frequent finding was related to the use of blacklisted calls (BLACKLIST), CWE-330: Use of Insufficiently Random Values. This is a critical security finding, as it can impact confidentiality leaks or bypass protection mechanisms

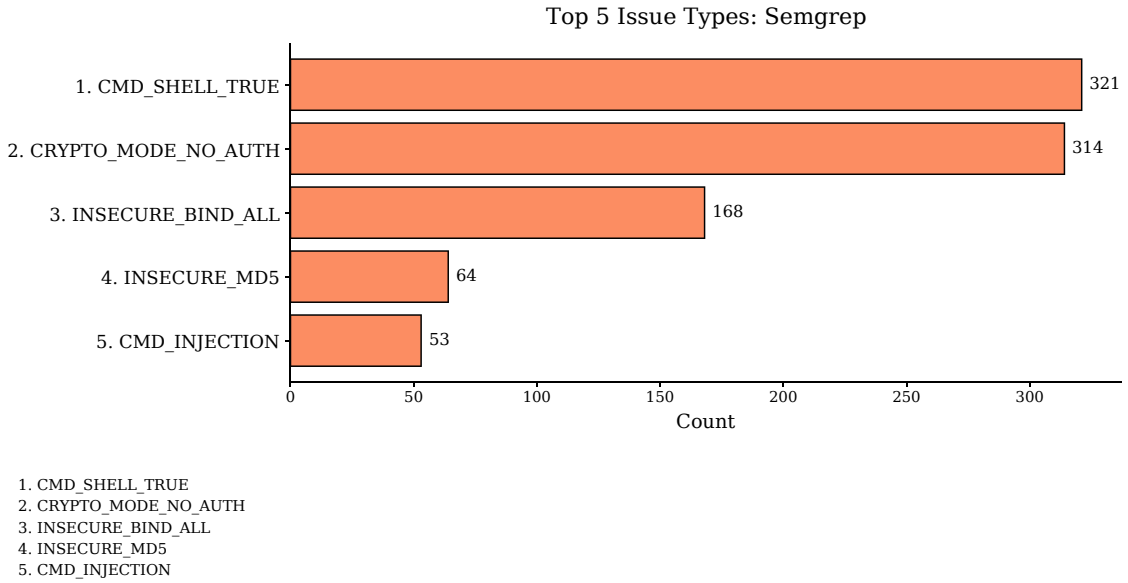



Figure 6.9: Top 5 Most Frequent Issues Detected by Semgrep

**Semgrep:** The pattern-matching engine Semgrep reported a much smaller, more focused set of high-confidence security issues. Its top finding, similar to Bandit, was CMD_SHELL_TRUE (338 instances), confirming the prevalence of insecure subprocess calls in the translated code.

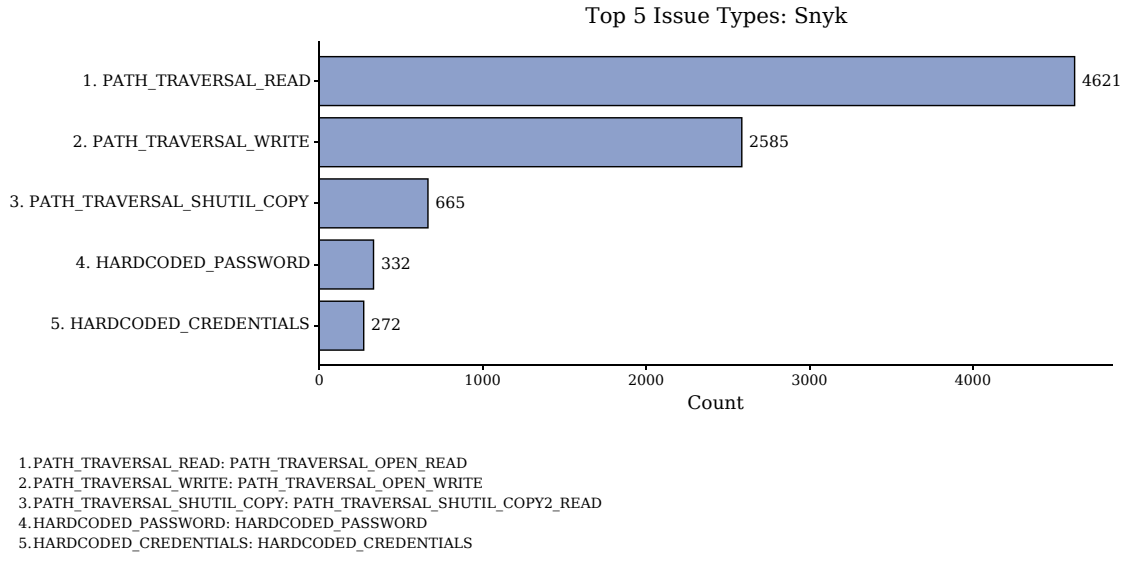



Figure 6.10: Top 5 Most Frequent Issues Detected by Snyk

**Snyk:** The commercial-grade tool Snyk identified command injection Path Traversals as the most frequent vulnerability, reporting 6,108 instances. Its findings demonstrate a strong focus on taint analysis, tracing how user-controllable input flows into potentially dangerous functions.

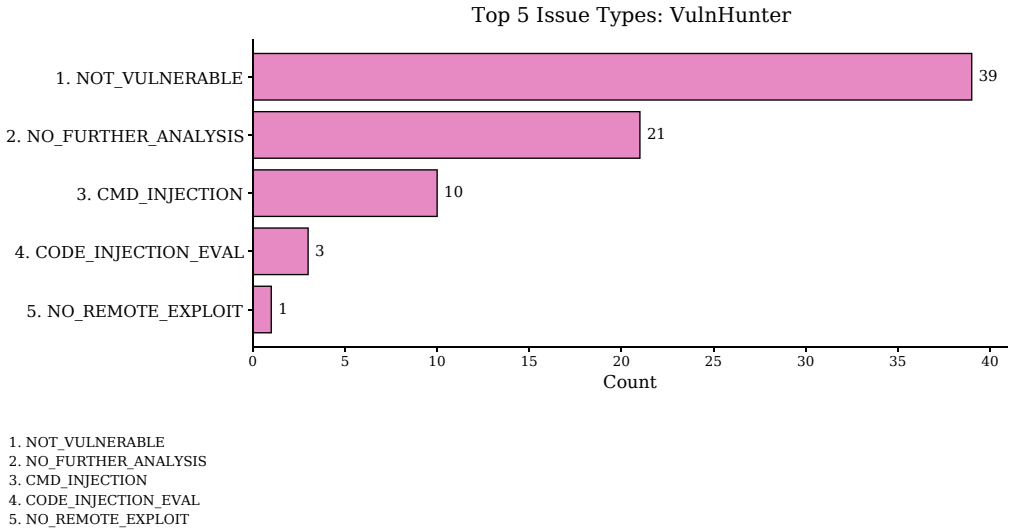



Figure 6.11: Top 5 Most Frequent Issues Detected by Vulnhuntr

**Vulnhuntr:** The hybrid tool Vulnhuntr, augmented by the Gemini LLM, provided a unique profile of results. Its most common output was NO_REMOTE_EXPLOIT (626 instances).

### 6.3.2 Cross-Tool Comparative Analysis

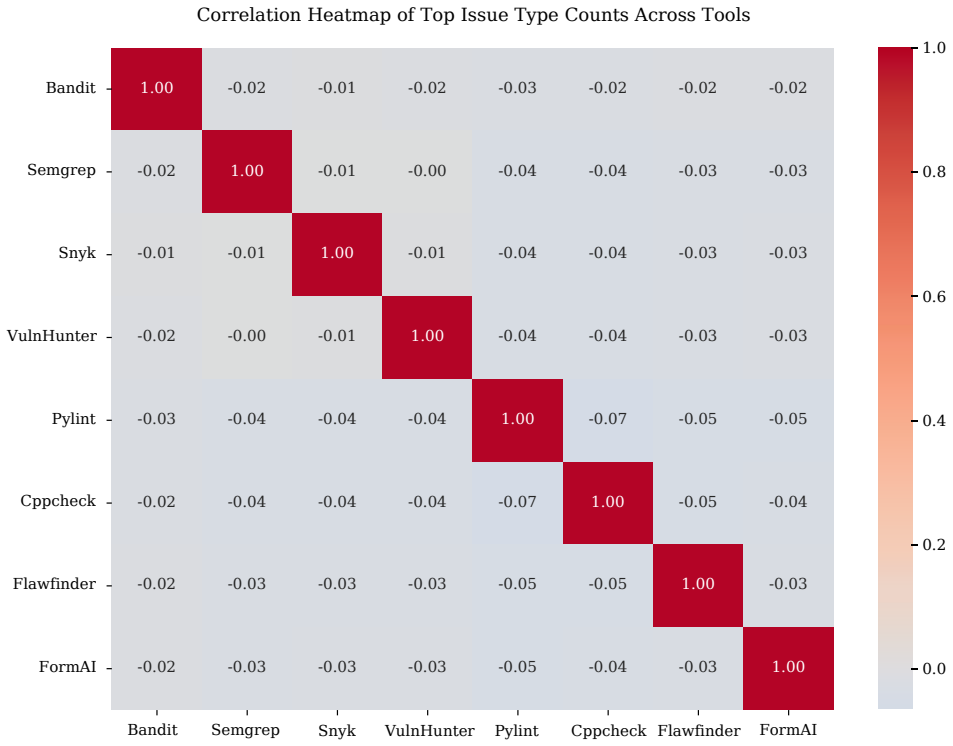



Figure 6.12: Correlation Heatmap of Top Issue Type Counts Across Tools

To understand the relationships and overlaps between the various analysis tools, a correlation matrix was generated based on the counts of the top 20 most frequent, standardized issue types found in the Python and C codebases.

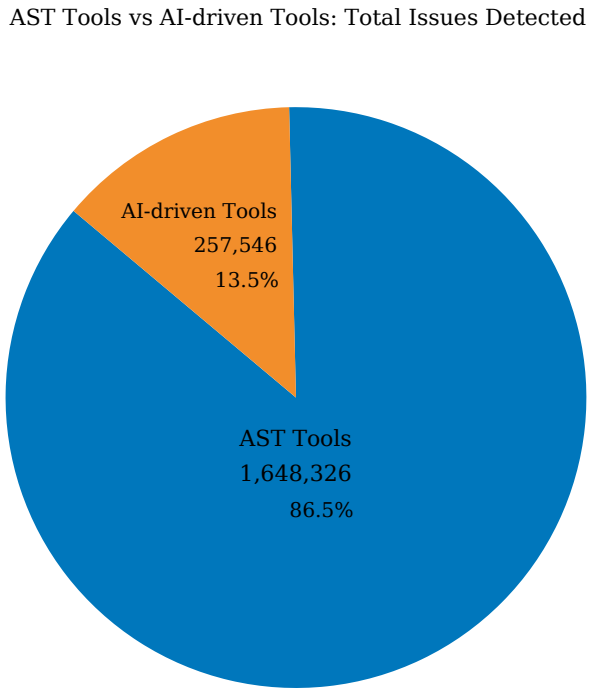



Figure 6.13: Total Issues Detected: AST-based vs. AI-driven Tools

A comparison between AST-based tools (Bandit, Semgrep, Cppcheck) and AI-driven tools (Snyk) based on issue severity provides further context. As seen in Figure 6.13, while the AST-based tools identified a significantly higher total volume of issues (1,648,326), the AI-driven tools demonstrated a greater capability in identifying and categorizing high-severity vulnerabilities.

## 6.4 Baseline Security Analysis of the FormAI V2 Dataset

By using earlier translation method, Gemini 2.0 API was used again to translate the FormAI v2 dataset to python snippets. After that, Bandit, Semgrep, Snyk and Vulnhuntr was used to generate security results. Pylint was not used as it is a linter based security tool which is not related to code security. Here are the general keywords that are used to categorize in both LLM output and SAST results:

- SQL Injection: sql injection, sql query, database query
- Command Injection: command injection, subprocess, shell=true, popen, os.system, subprocess call, shell process, tainted env args, user controlled data,

malicious actor, execute commands, shell command, dangerous subprocess, system call

- Cross-Site Scripting (XSS): cross-site scripting, xss
- Path Traversal: path traversal, directory traversal, file path injection, file path
- Buffer Overflow: buffer overflow
- Integer Overflow: integer overflow, integer underflow
- Denial of Service: redos, regular expression denial of service, denial of service, race condition, dos, large volume of data
- Insecure Cryptography: md5, sha1, insecure hash, crypto, ssl, tls, protocol sslv23, encryption mode, weak randomness, hash algorithm, cryptographic signature, aead mode, message authentication, gcm, deprecated, pycrypto, pseudo-random, random, pseudo-random generators, security/cryptographic purposes, b311
- Insecure Network Configuration: bind, all interfaces, 0.0.0.0, socket, network interfaces, unintended traffic, bind to all, publicly expose, available interfaces
- Insecure Deserialization: insecure deserialization, deserialization
- Insecure Direct Object Reference: insecure direct object reference, idor
- Cross-Site Request Forgery: cross-site request forgery, csrf
- Insecure File Permissions: insecure file permissions, 0o666, 0o777, 0o755, permissive, grant access, file permissions, widely permissive
- XML External Entity (XXE): xxe, xml external entity, xml library, defusedxml, xml bomb
- Server-Side Request Forgery (SSRF): ssrf, server-side request forgery, internal service, remote entry point
- Code Injection: code injection, format string, shell command injection, rce, remote code execution, command string, f-string
- Input Validation: input validation, unvalidated user input, user input, input sanitization, sanitize, validation
- Error Handling: error handling, try except pass, exception handling
- Deprecated Libraries: deprecated, no longer actively maintained, pycrypto, outdated library
- Code Quality & Style: pep 8, docstring, line too long, invalid name, hardcoded tmp directory, temp file, code quality
- Miscellaneous Security Finding: security, vulnerability, insecure, memory leak, remote entry point, network entry point, malicious, attack

An overall summary of the findings across all tools is presented below.

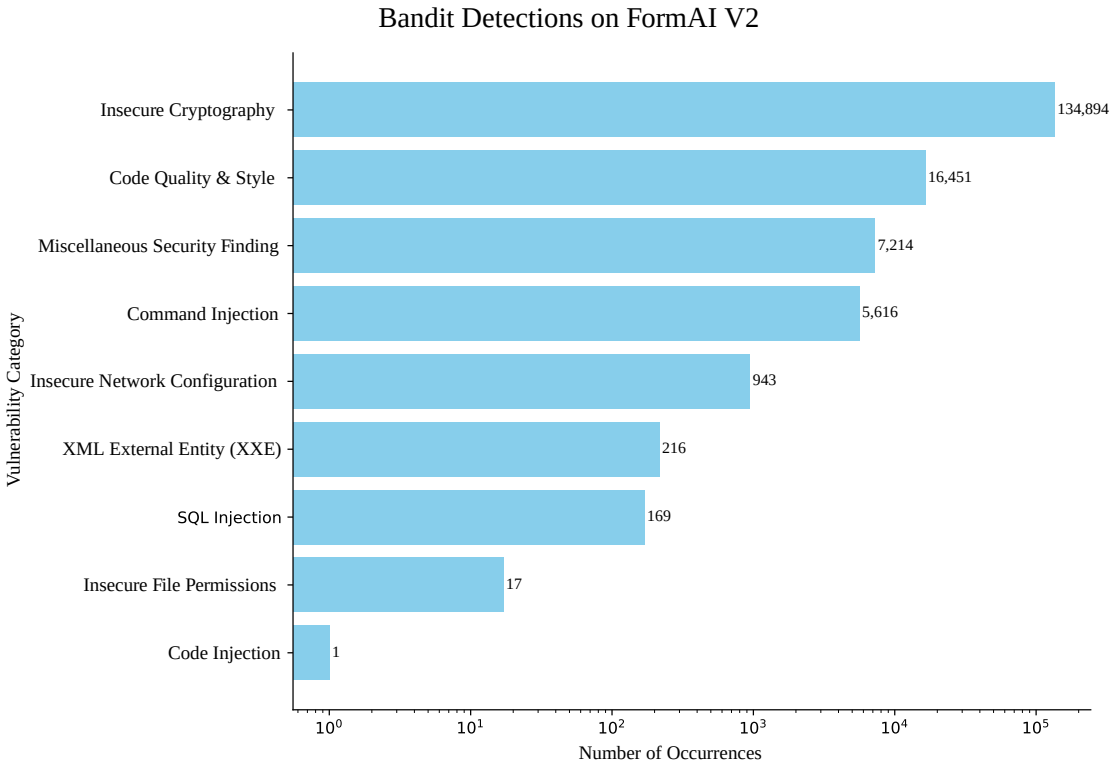



Figure 6.14: Total Issues Detected on FormAI V2 by Bandit

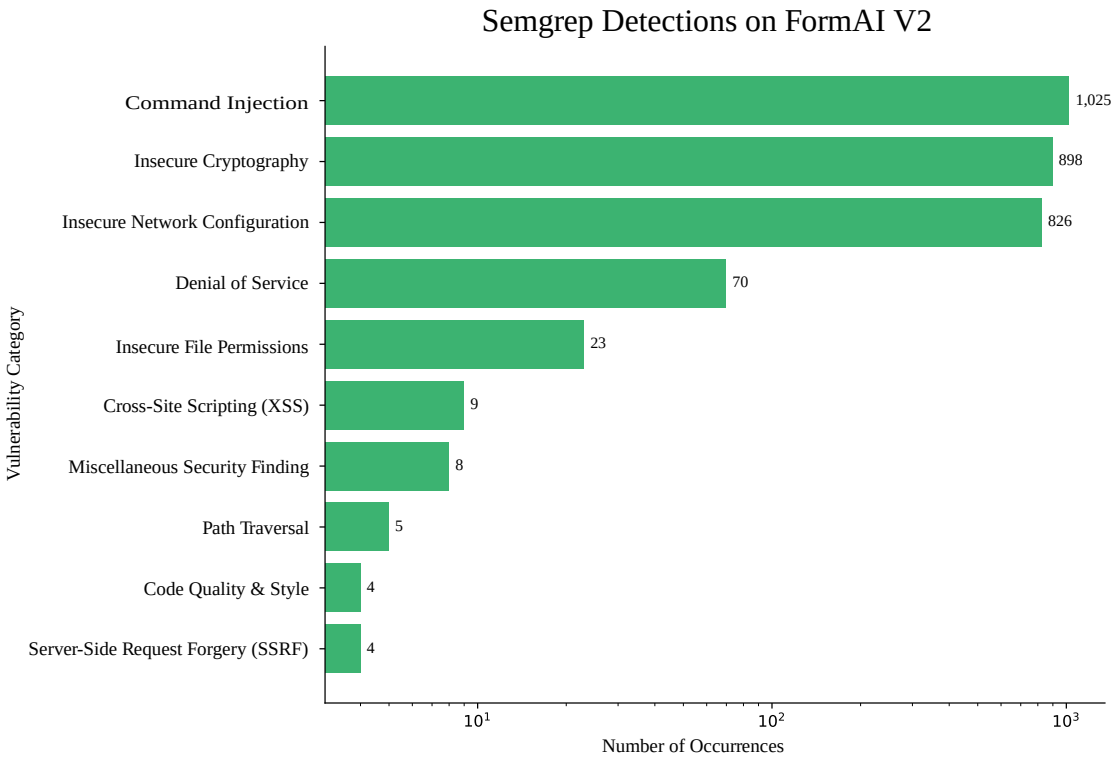



Figure 6.15: Total Issues Detected on FormAI V2 by Semgrep

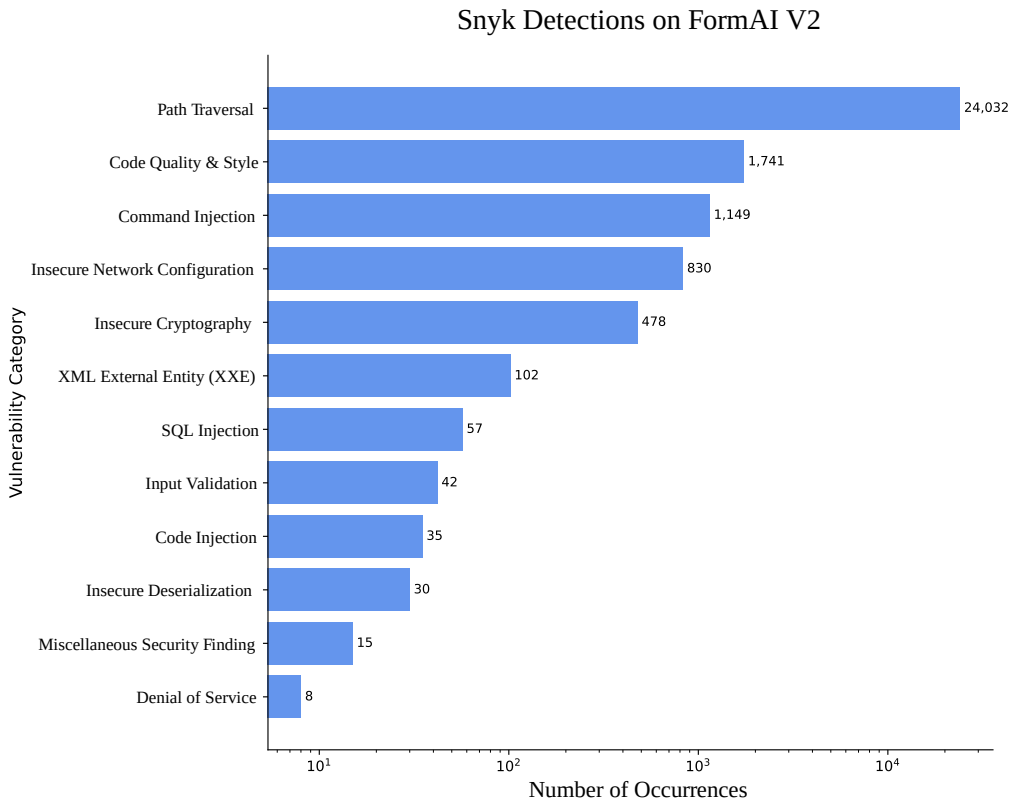



Figure 6.16: Total Issues Detected on FormAI V2 by Snyk

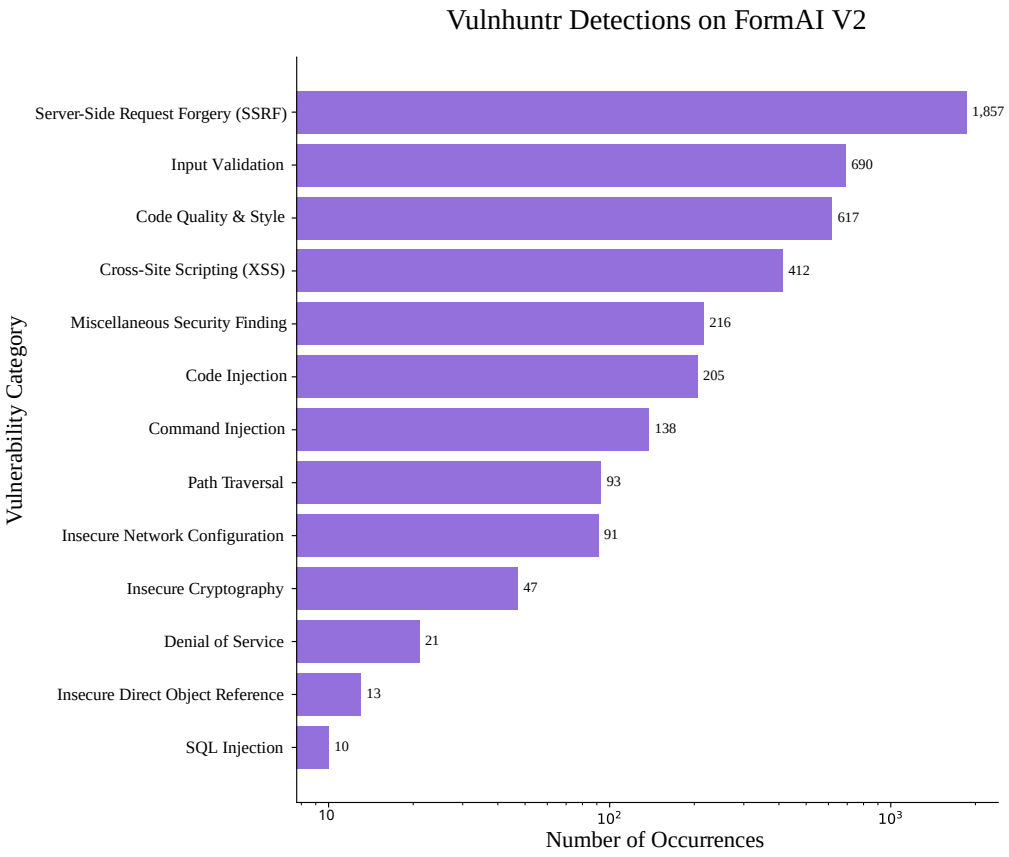



Figure 6.17: Total Issues Detected on FormAI V2 by Vulnhuntr

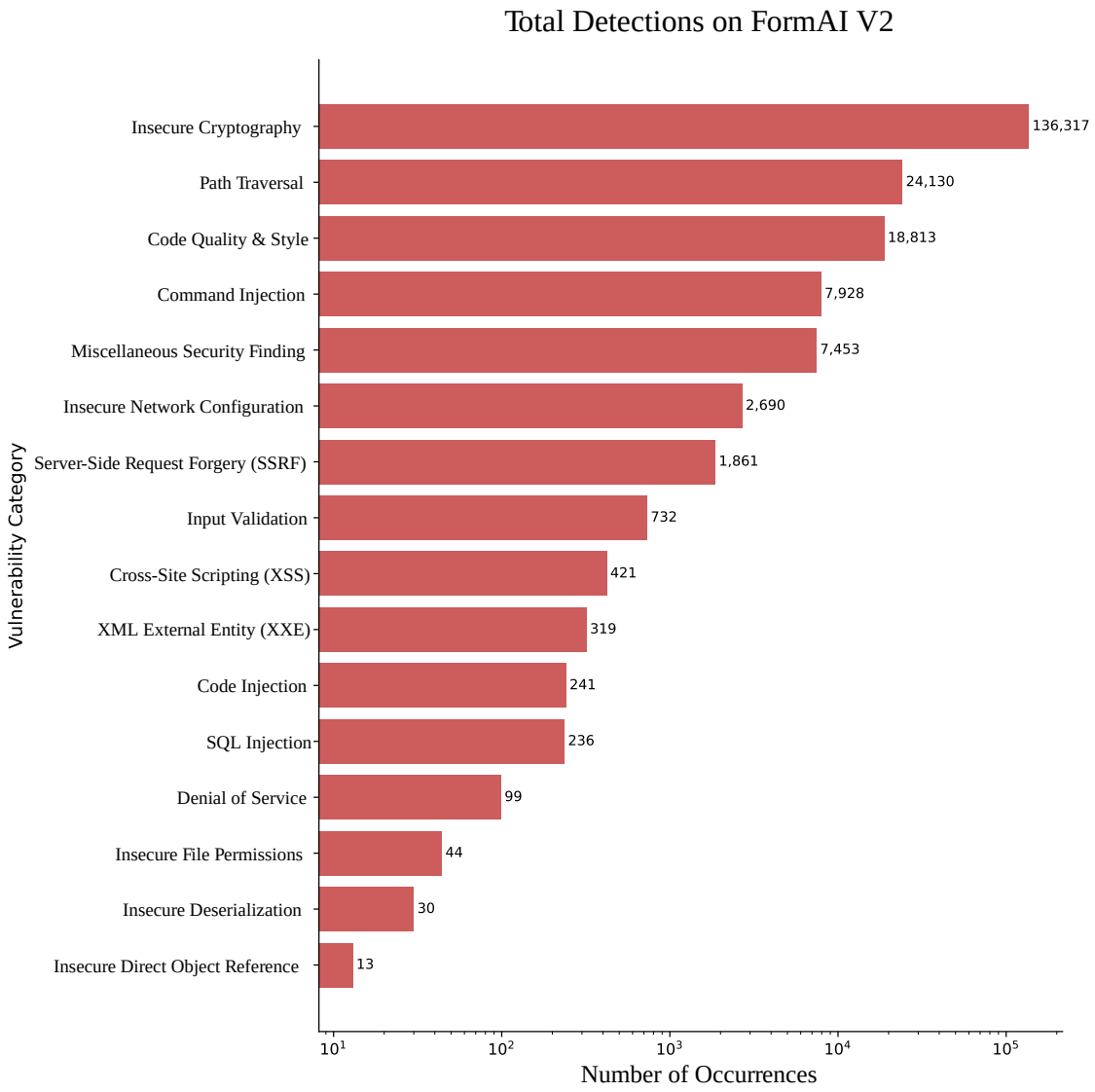



Figure 6.18: Issues Detected on FormAI V2

**Bandit:** Bandit has detected over 130000 Insecure cryptography issues, 16000 Code quality and style issues and 5000 Command injection vulnerabilities. There are also around 7200 miscellaneous findings. Bandit also reported several XML external entity, insecure network configuration, SQL injections as well as some insecure file permissions.

**Semgrep:** Semgrep has detected around 1000 command injections. It also detected around 800 Insecure cryptography, Insecure Network Configuration. There are several DoS (Denial of service) attacks mentiond and some insecure file permissions. It also reported 9 Cross-site Scripting (XSS). And very few path traversal, code quality and server-side request forgery

**Snyk:** Snyk has reported approximately 24000 path traversal issues in the codes, 1700 code quality and style issues, 1100 command injection probelems which are often shell related, it has 800 Insecure network configuration and 400 Insecure cryptography. It also reported 102 XML external entity reports, some SQL injections,

input validations and miscellaneous security findings.

**Vulnhuntr:** Vulnhuntr this time was also powered by Gemini 2.0 Flash model. Vulnhuntr caught 1857 Server-side request forgery issues, 690 input validation issues, 617 Code quality and style related issues. There are also mention of Cross-Site Scripting (XSS) and several Miscellaneous Security Finding, Code Injections, Command Injections, Path Traversal, Insecure Netowrk Configuration, Insecure Cryptography nad very Denial of service SQL injections.

**FormAI V2 Python Ground Truth:** The ground truth is built on the combination of all the 4 SAST tool results. Results generated by the models on FormAI V2 will be compared to this ground truth results to evaluate their performance and scores accross the board.

## 6.5 The SAST-Informed Training Dataset (Dataset 1)

The initial fine-tuning phase aimed to teach the LLMs to identify vulnerabilities similar to a SAST tool. To achieve this, the first training dataset was constructed using the vulnerability reports generated by the SAST analysis of the FormAI v1 Python codebase. The dataset consists of approximately 40503 training examples. Below is an example of what each of the example looks like:


```
{"instruction": "Analyze the provided Python code and identify security
vulnerabilities.", "input": "        filename = input(\"Enter filename:
\")\n        try:\n            with open(filename, 'w', newline='') as
csvfile:\n                writer = csv.writer(csvfile)\n
for student in self.students:\n                    writer.writerow([
student.id, student.name, student.score])\n            print(\"
Database saved to file.\")\n", "output": "Vulnerability detected by
snyk: Unsanitized input from user input flows into an unquoted
CSV-writer (csv.writer). Use quoting=csv.QUOTE_ALL when the input
is flowing from an untrusted source."}
```


This approach was designed to directly train the models on the patterns that existing, specialized tools recognize as insecure, with the goal of embedding this knowledge into the LLMs themselves.

## 6.6 Performance of Models Trained on Dataset 1

The three fine-tuned models (Llama 7B, Deepseek 6.7B, Gemma 7B), along with the Gemini 2.5 Pro model via API calls (zero-shot), were evaluated on a test set derived from the FormAI v1 dataset. Every output follows this format:

```
"filepath,model_analysis"
```

Due to hardware limitations, this research audited a random sample of 5000 out of FormAI V1 and V2 each. Here is a sample of each models output for one file:

- **Llama:** FormAI 57079.py,"This code is a Python implementation of a scientific calculator that allows users to perform basic arithmetic operations and calculate the square root, factorial, and power of a number. The code is vulnerable to several security issues, including: 1. Insecure Direct Object Reference (IDOR): The code uses a global variable 'memory' to store the value of the memory, which can be modified by any user. This allows an attacker to modify the value of the memory, potentially allowing them to gain unauthorized access to sensitive information or perform malicious actions. 2. Insecure Direct Memory Access (IDMA): The code uses the 'memory' variable to store the value of the memory, which can be modified by any user. This allows an attacker to modify the value of the memory, potentially allowing them to gain unauthorized access to sensitive information or perform malicious actions. 3. Insecure Deserialization: The code uses the 'eval' function to evaluate user input, which can be used to execute arbitrary Python code. This allows an attacker to execute arbitrary Python code, potentially allowing them to gain unauthorized access to sensitive information or perform malicious actions. 4. Insecure Libraries"

- **Deepseek Coder:** "FormAI 48786.py","The provided python script does not have any major security vulnerability as it doesn't involve handling user inputs or performing operations that could be exploited by malicious users. However, there might be some minor issues related to coding style which can be improved such as proper indentation and use of comments where necessary. Also, if this was part of larger system, care should also be taken about potential buffer overflows when dealing with large amounts of data. The current program allocates space but never checks whether these spaces get overwritten after they become unnecessary due to lack of references. This may lead to unintentional leaks of sensitive information. It would be better to explicitly set values to zero once their usage ends. Overall, while this isn't directly a vulnerability per se, best practices still apply here."

- **Gemma:** "FormAI 97067.py","- **Insecure Input Validation:** The code does not validate the input length of 'str input' and 'char to count', which could lead to potential buffer overflows and code injection vulnerabilities. - **Character Comparison:** The code compares the character 'char' in the input string 'str input' with the character 'char to count' directly, which is susceptible to character substitution vulnerabilities if the input character is modified."

- **Gemini 2.5 API:** FormAI 37132.py,No vulnerabilities found.

### 6.6.1 Quantitative Results

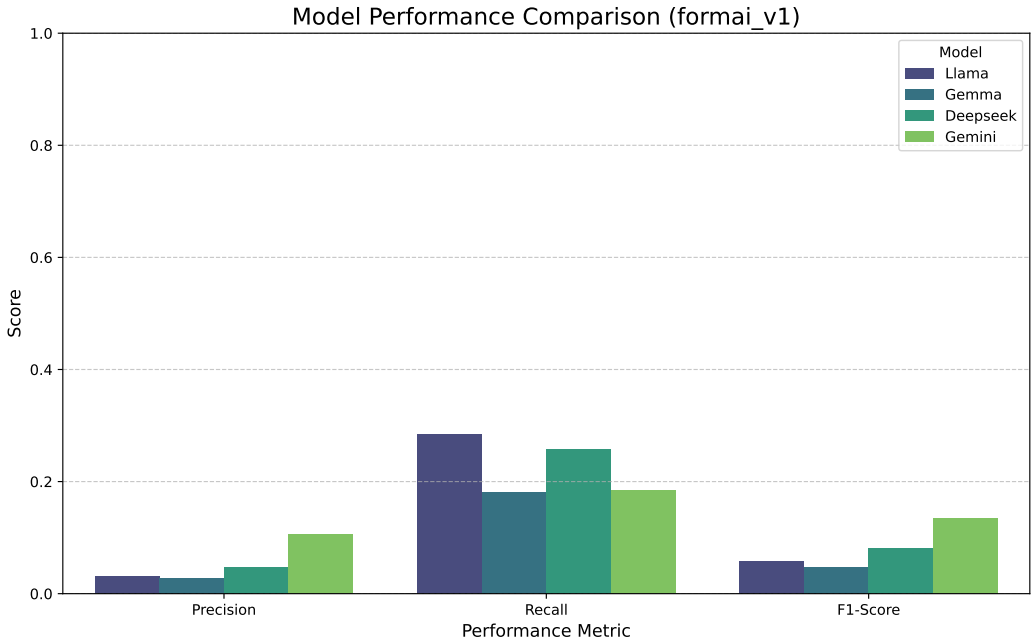



Figure 6.19: Model Comparison on First run

| Model | Precision | Recall | F1-Score |
|---|---|---|---|
| Llama | 0.0320627 | **0.284131** | 0.0576229 |
| Gemma | 0.0277888 | 0.180645 | 0.0481679 |
| Deepseek | 0.0481446 | 0.258464 | 0.0811695 |
| Gemini | **0.106988** | 0.184962 | **0.135562** |

Table 6.1: Model Scores on First run on FormAI_V1

The overall performance is quite poor, considering there were no optimizations done. As shown in the comparison chart, no single model excelled across all metrics.

The relatively low scores across the board suggest that while the models learned to recognize some patterns, training on isolated, SAST-generated snippets was not sufficient for vulnerability detection.

### 6.6.2 Confusion Matrices

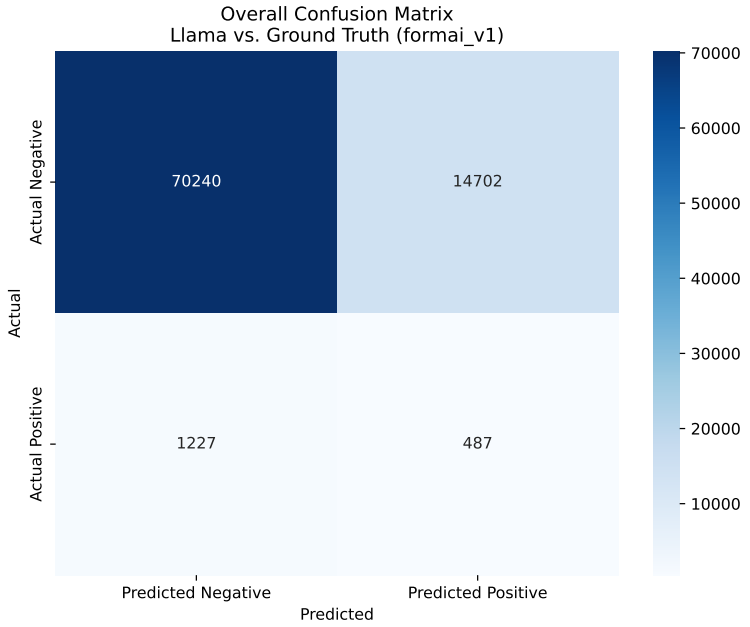



Figure 6.20: Llama confusion matrix on First run

**Llama:** Llama has successfully identified 487 issues while misidentifying more than 15000 issues.

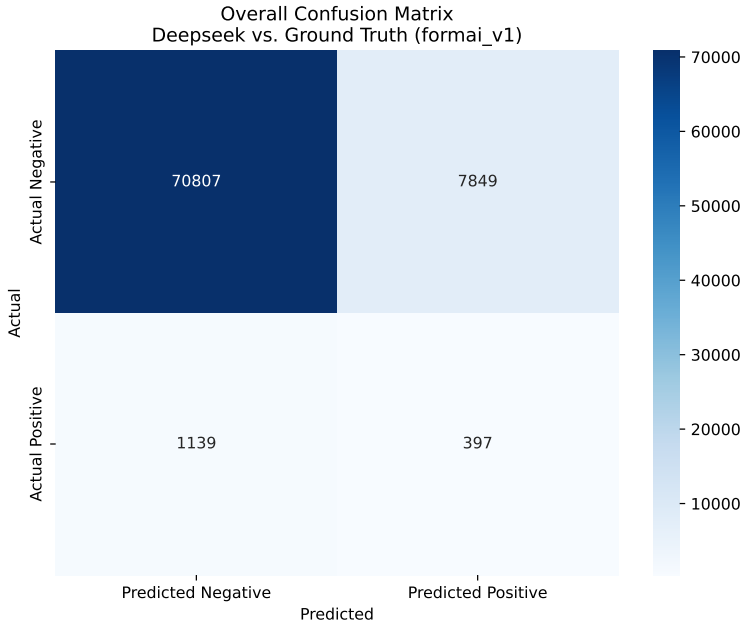



Figure 6.21: Deepseek confusion matrix on First run

**Deepseek:** Deepseek has successfully identified 397 issues while misidentifying more than 8000 issues.

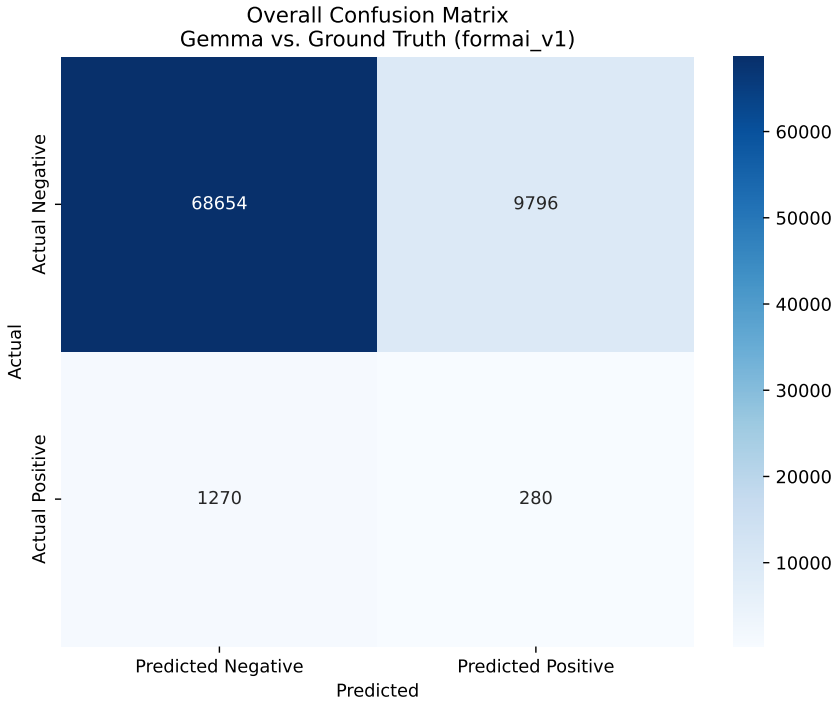



Figure 6.22: Gemma confusion matrix on First run

**Gemma:** Gemma has successfully identified 280 issues while misidentifying more than 10000 issues.

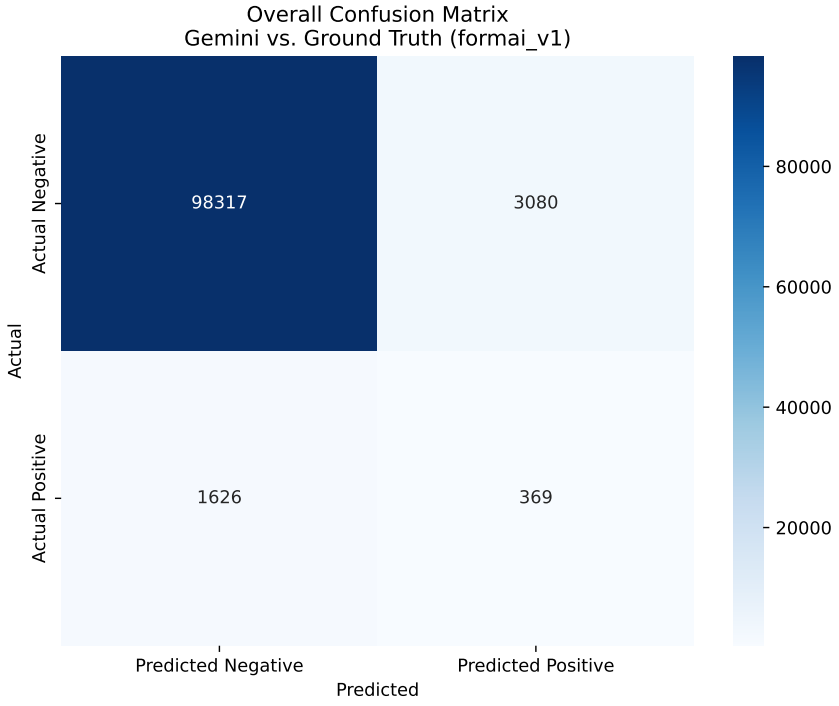



Figure 6.23: Gemini confusion matrix on First run

**Gemini:** Gemini has successfully identified 369 issues while misidentifying more

than 4500 issues.

## 6.7 Performance of Models on Unseen Data (FormAI v2)

To assess more accurate state of the models, the models were then run on FormAI V2, which was not included in their training. This would demonstrate how the model would behave with unseen codes.

### 6.7.1 Quantitative Results

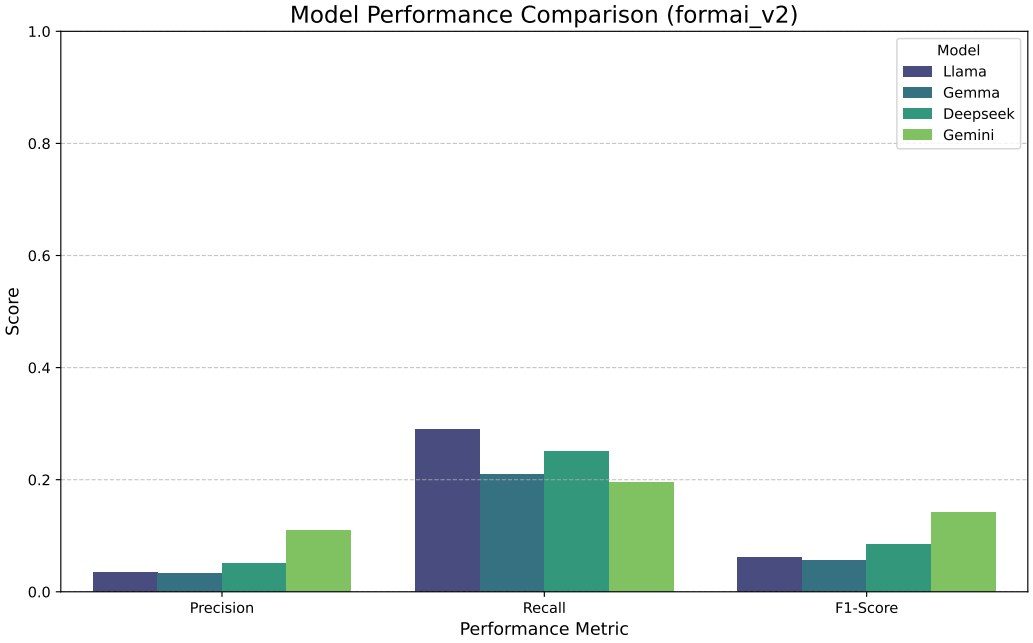



Figure 6.24: Model Comparison on V2, First run

| Model | Precision | Recall | F1-Score |
|---|---|---|---|
| Llama | 0.0349635 | **0.289692** | 0.0623963 |
| Gemma | 0.0328273 | 0.209302 | 0.0567533 |
| Deepseek | 0.0508616 | 0.250585 | 0.0845599 |
| Gemini | **0.110123** | 0.195864 | **0.140981** |

Table 6.2: Model Scores on V2, First run

The overall performance is mixed, considering these codes were not in the training dataset. As shown in the comparison chart, no single model was sometimes better in all metrics.

The relatively low scores across the board suggest that while the models learned to recognize some patterns, training on isolated, SAST-generated snippets was not sufficient for vulnerability detection.

### 6.7.2 Confusion Matrices

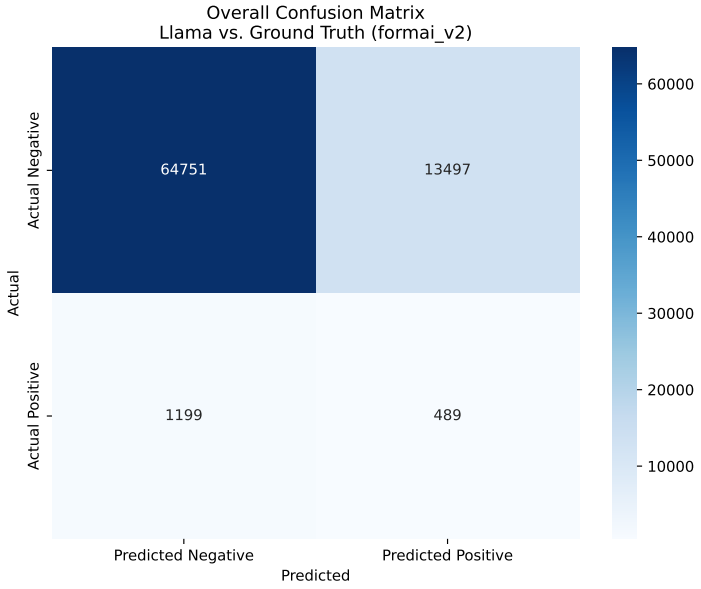



Figure 6.25: Llama confusion matrix on V2 First run

**Llama:** Llama has successfully identified 489 issues while misidentifying more than 14000 issues.

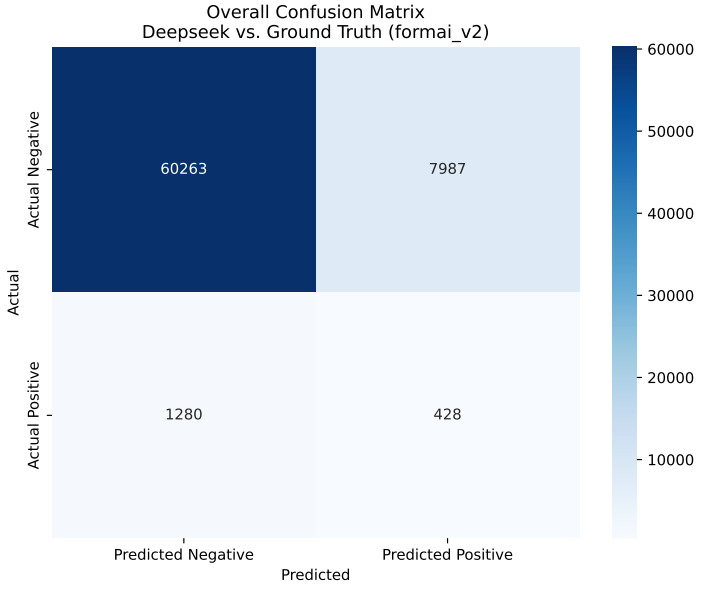



Figure 6.26: Deepseek confusion matrix on V2 First run

**Deepseek:** Deepseek has successfully identified 428 issues while misidentifying more than 9000 issues.

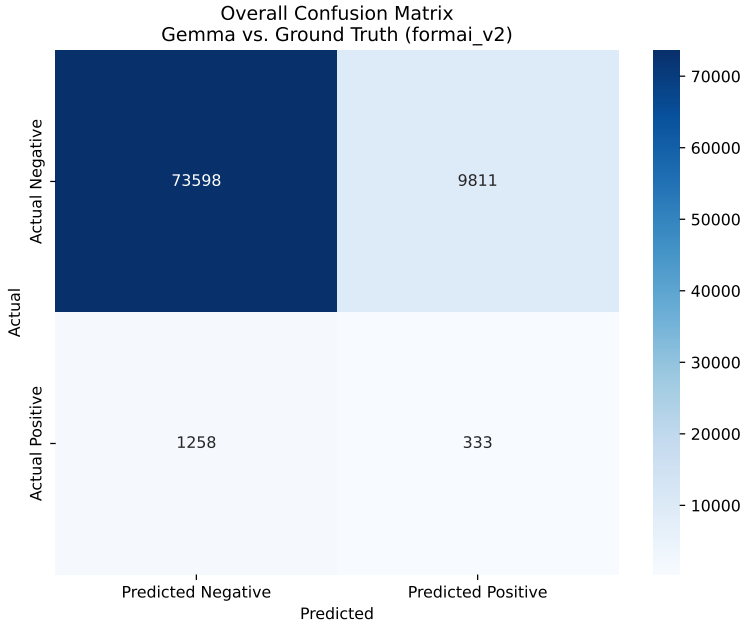



Figure 6.27: Gemma confusion matrix on V2 First run

**Gemma:** Gemma has successfully identified 333 issues while misidentifying more than 11000 issues.

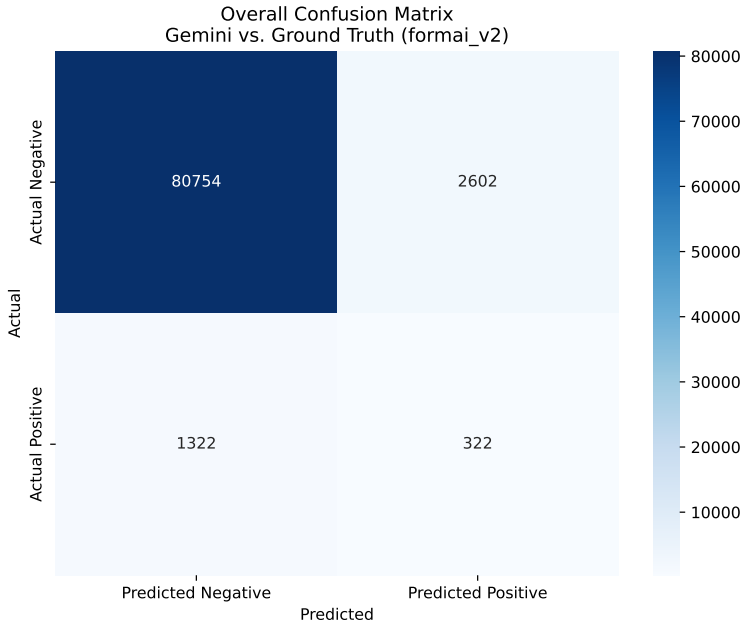



Figure 6.28: Gemini confusion matrix on V2 First run

**Gemini:** Gemini has successfully identified 322 issues while misidentifying more than 3700 issues.

## 6.8 Performance of optimized Models Trained on Dataset 1

After initial run, the training dataset was kept the same but to optimize the models, the training arguments and the hyperparameters were needed to be changed. After several iterations, an optimal settings was decided. There were also prompt engineering to filter out unnecessary outputs generated by both the LLMs and the API. As a result, a significant boost in scores were noticed.

### 6.8.1 Quantitative Results

The overall performance increased significantly after optimization. However, the precision remains poor overall. But the recalls of the LLMs jumped by a large margin. The conclusion is, the dataset must contain secure code snippets so that the LLM know which codes look secure. But due to hardware limitation this could not be achieved.

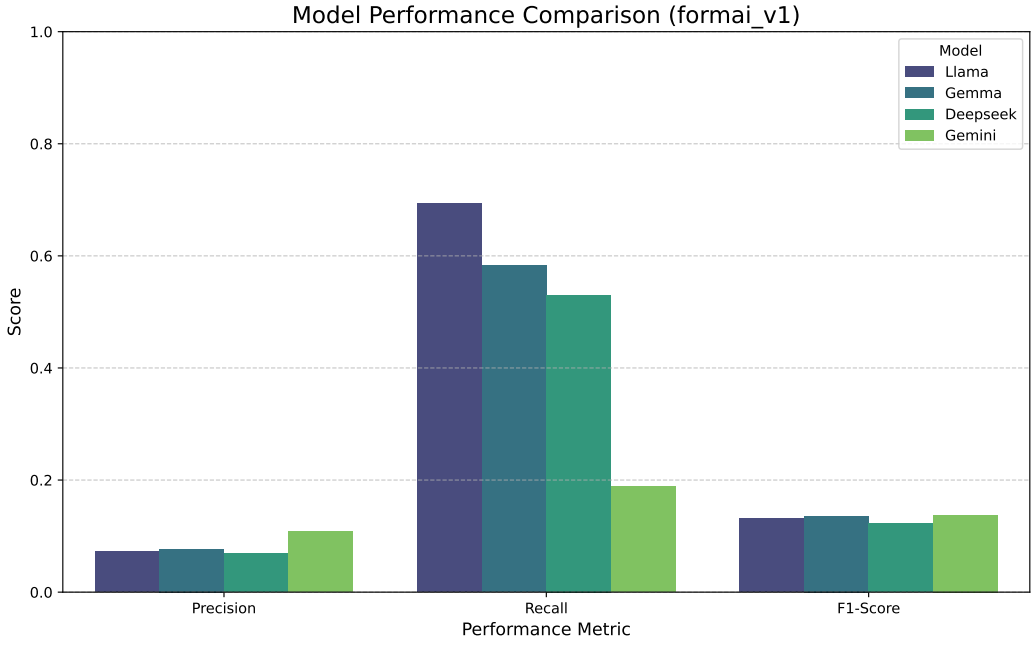



Figure 6.29: Model Comparison on Optimized run on FormAI_v1

| Model | Precision | Recall | F1-Score |
|---|---|---|---|
| Llama | 0.0725782 | **0.693162** | 0.131398 |
| Gemma | 0.0770149 | 0.582581 | 0.136045 |
| Deepseek | 0.0693864 | 0.530333 | 0.122717 |
| Gemini | **0.108319** | 0.188253 | **0.137514** |

Table 6.3: Model Scores on Optimized run on FormAI_v1

The relatively higher recall suggest that while the models learned to identify vulnerabilities but the precision suggests the models need to learn to detect secure codes as well.

### 6.8.2 Confusion Matrices

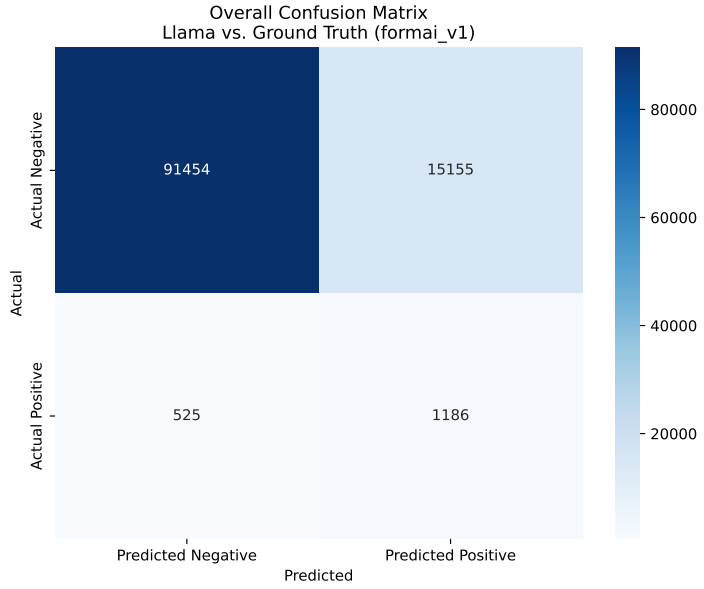



Figure 6.30: Llama confusion matrix on Optimized run

**Llama:** Llama has successfully identified 1186 issues while misidentifying more than 15500 issues.

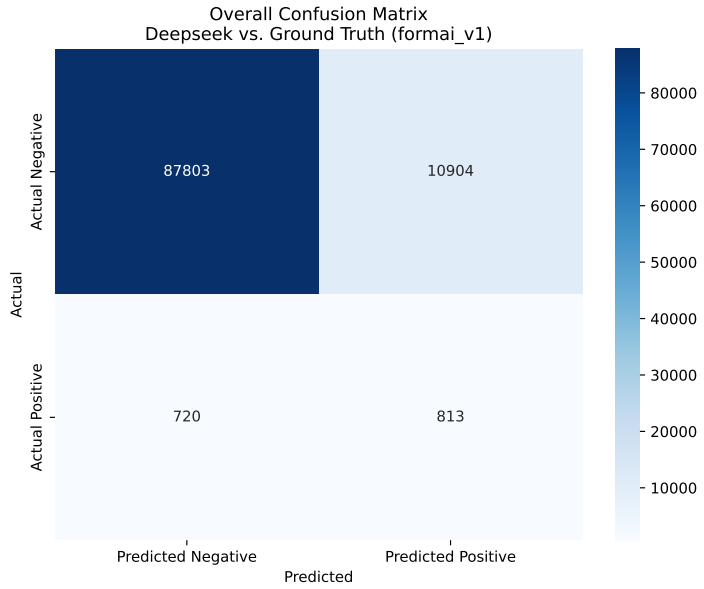



Figure 6.31: Deepseek confusion matrix on Optimized run

**Deepseek:** Deepseek has successfully identified 813 issues while misidentifying more than 11000 issues.

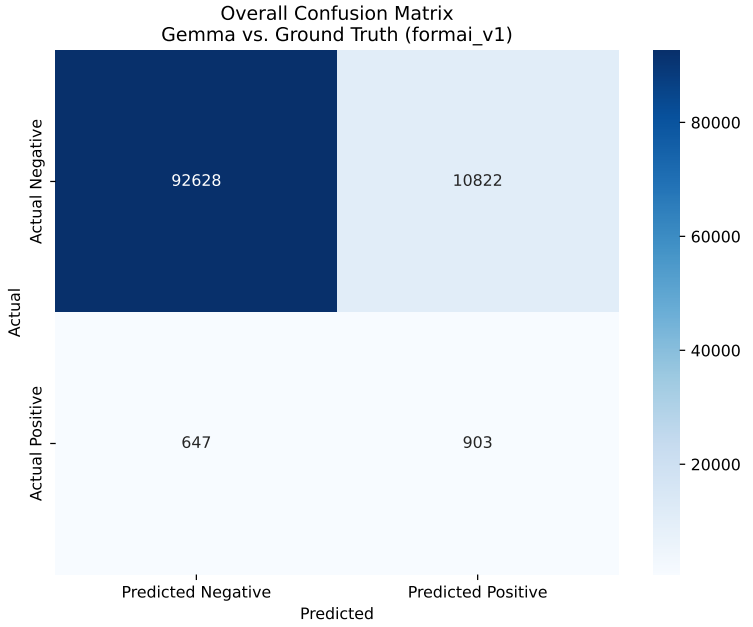



Figure 6.32: Gemma confusion matrix on Optimized run

**Gemma:** Gemma has successfully identified 903 issues while misidentifying more than 11000 issues.

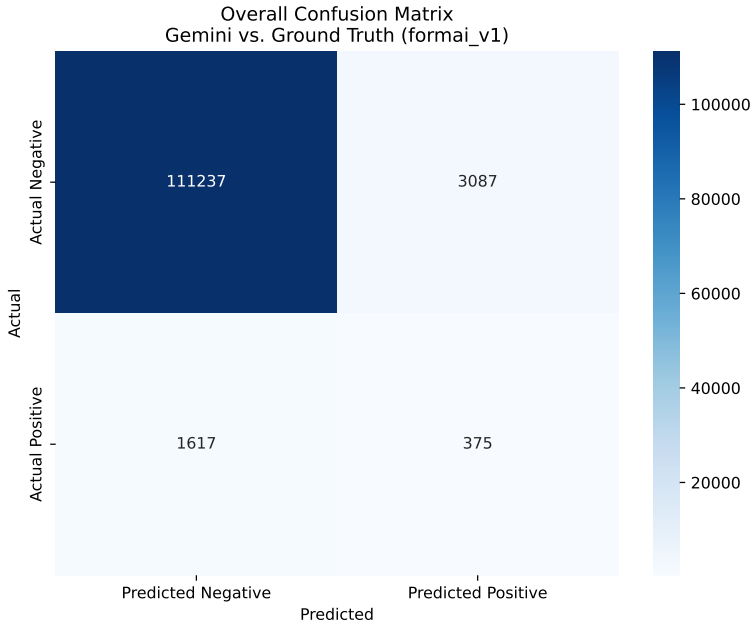



Figure 6.33: Gemini confusion matrix on Optimized run

**Gemini:** Gemini has successfully identified 375 issues while misidentifying more than 3100 issues.

## 6.9 Performance of Optimized Models on Unseen Data (FormAI v2)

Now, evaluation of the optimized models on the unseen data will be conducted to see how the models perform on unseen dataset. It is expected that the results would be similar to optimized model audits of the V1 dataset.

### 6.9.1 Quantitative Results

The overall performance is mixed, considering these codes were not in the training dataset. As shown in the comparison chart, no single model was sometimes better in all metrics.

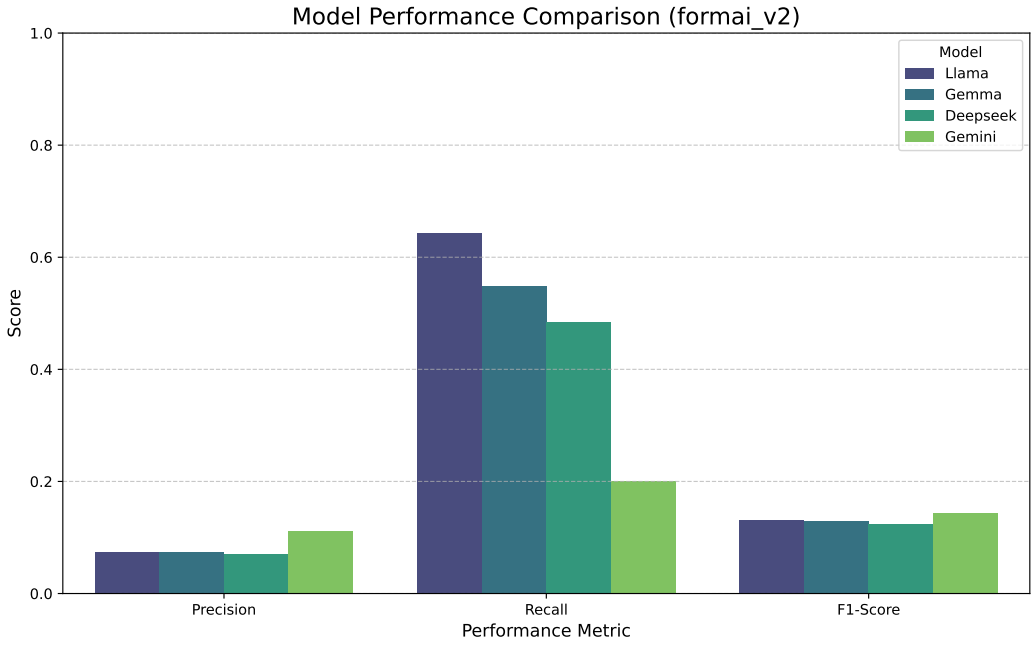



Figure 6.34: Model Comparison on V2, Optimized run on FormAI_V2

| Model | Precision | Recall | F1-Score |
|---|---|---|---|
| Llama | 0.0733957 | **0.643447** | 0.131762 |
| Gemma | 0.0731544 | 0.548246 | 0.129085 |
| Deepseek | 0.0705296 | 0.485183 | 0.123156 |
| Gemini | **0.111789** | 0.200121 | **0.143447** |

Table 6.4: Model Comparison on V2, Optimized run on FormAI_V2

The relatively low scores across the board suggest that while the models learned to recognize some patterns, training on isolated, SAST-generated snippets was not sufficient for vulnerability detection.

### 6.9.2 Confusion Matrices

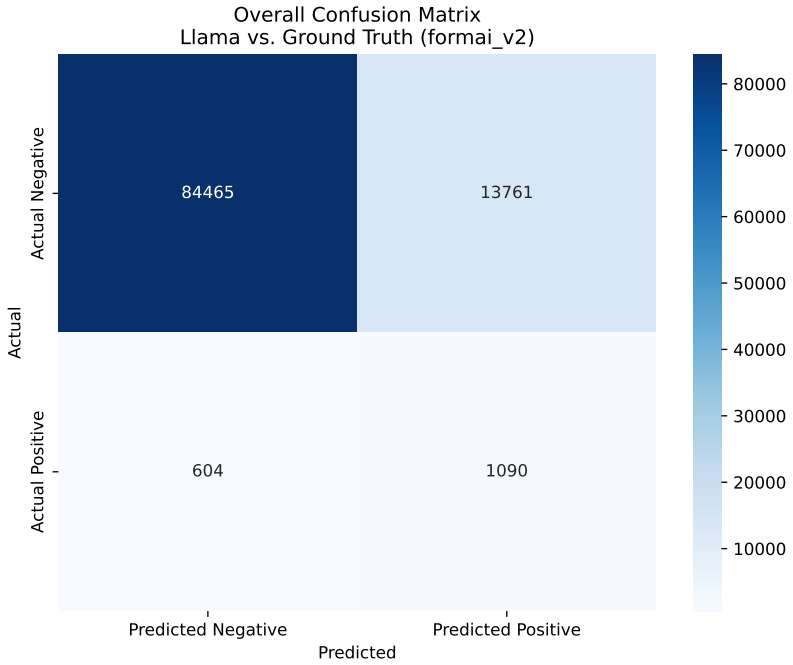



Figure 6.35: Llama confusion matrix on V2 Optimized run

**Llama:** Llama has successfully identified 1090 issues while misidentifying more than 14000 issues.

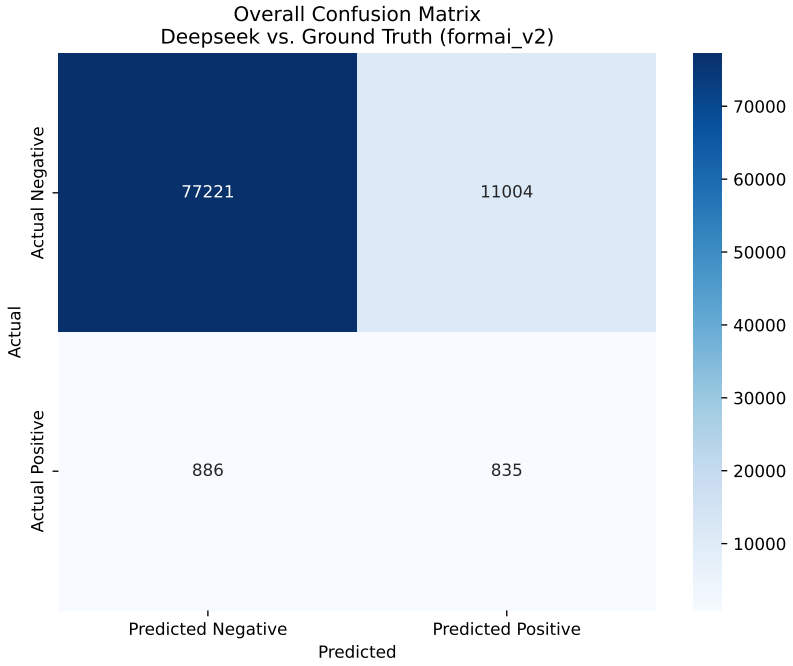



Figure 6.36: Deepseek confusion matrix on V2 Optimized run

**Deepseek:** Deepseek has successfully identified 835 issues while misidentifying more than 11000 issues.

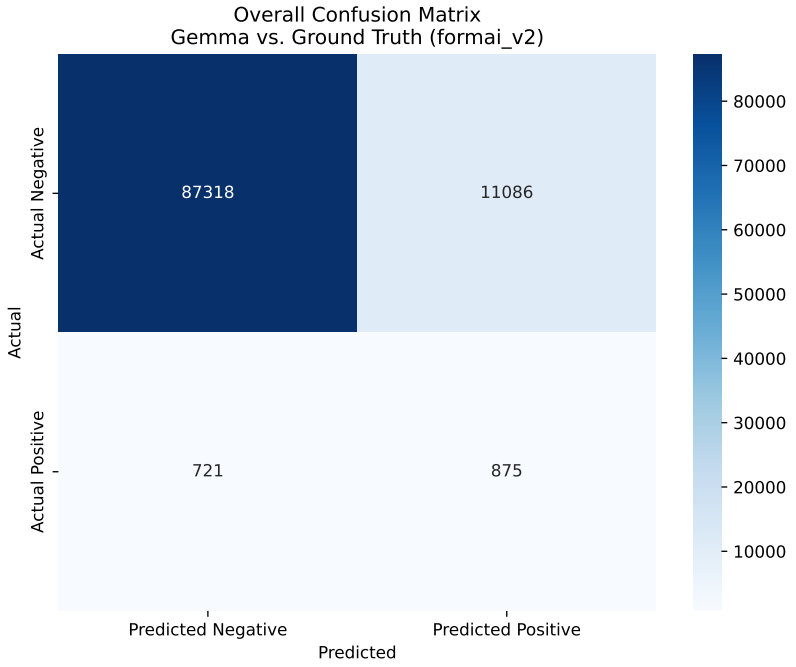



Figure 6.37: Gemma confusion matrix on V2 Optimized run

**Gemma:** Gemma has successfully identified 837 issues while misidentifying more than 11000 issues.

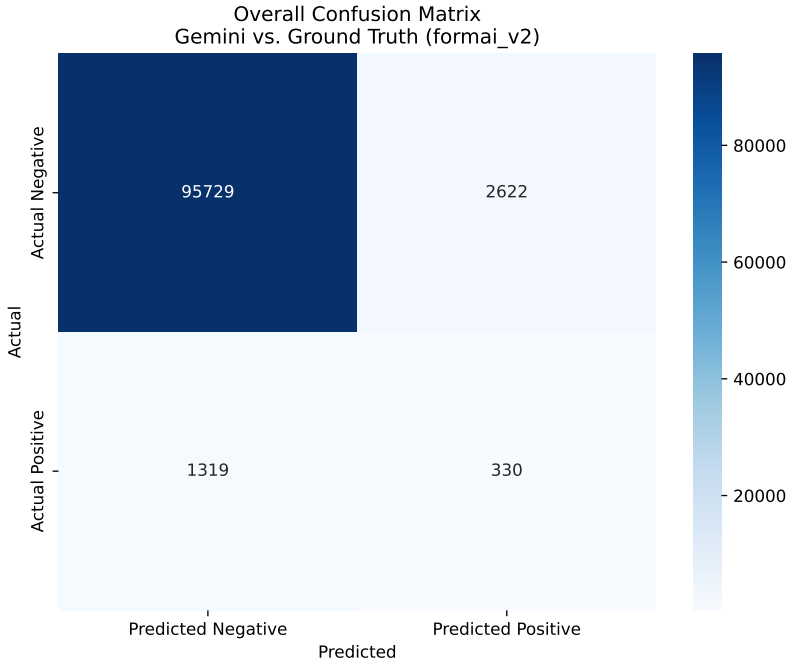



Figure 6.38: Gemini confusion matrix on V2 Optimized run

**Gemini:** Gemini has successfully identified 330 issues while misidentifying more than 3800 issues.

## 6.10 The Full-Code Security Classification Dataset (Dataset 2)

The suboptimal performance and qualitative issues observed in Phase 1 highlighted a key limitation: analyzing just the code snippets context necessary for accurate vulnerability detection. To address this, a second was developed that included the entirety of a python program, and from previously categorized framework, the models were trained to output if a code is either secure or vulnerable. If vulnerable, it must say the category or categories the vulnerabilities fall on.

The dataset consists of 7,778 samples, half of which are secure and half of which are vulnerable. The initial plan to train with 90000 samples could not be achieved because of hardware's memory limitation. in order to fine tune the models with minimal input loss training dataset was reduced to 7,778 samples but still mean token count was used as input in order to fit the datasets. Below is a sample of what each line looks like in the training dataset:


```
{"text": "### INSTRUCTION:\nAnalyze the following Python code for
security vulnerabilities. If the code is secure, respond with the
single word 'secure'. If vulnerabilities are present, respond with
a comma-separated list of all vulnerability categories you have
identified.\n\n### CODE:\nimport random\nimport time\n\nNUM_CARDS
= 16\n\ncards = [''] * NUM_CARDS  # An array to store the cards
\ncards_visible = [False] * NUM_CARDS  # An array to store if the
cards are visible or not\nprevious_card = None  # A variable to
store the previous card picked\nprevious_card_index = -1  # A
variable to store the index of the previous card picked\n\n\n#
Function to initialize the cards array with values from 'A' to
'H'\ndef initialize_cards():\n    card_value = 'A'\n    for i
in range(0, NUM_CARDS, 2):\n        cards[i] = card_value\n
cards[i + 1] = card_value\n        card_value =
chr(ord(card_value) + 1)\n\n\n# Function to shuffle the cards
\ndef shuffle_cards():\n    random.seed(time.time())\n
for i in range(NUM_CARDS - 1, 0, -1):\n
j = random.randint(0, i)\n        cards[i], cards[j] = cards[j],
cards[i]\n\n\n# Function to print the cards\ndef print_cards():
print(\"  \", end=\"\")\n    for i in range(NUM_CARDS):\n
print(f\"{i:2}\", end=\"\")\n    print()\n\n
for i in range(NUM_CARDS):\n if i == 0:\n      print(chr(201),
end=\"\")\n elif i == NUM_CARDS - 1:\n     print(chr(187))\n
else:\n    print(chr(203) + chr(205), end=\"\")\n\n
print(end=\"\")  # Ensure there's no extra space after
printing characters\n\n for i in range(NUM_CARDS):\n if i == 0:
print(chr(186), end=\"\")\n        if cards_visible[i]:\n
print(f\"{cards[i]} \", end=\"\")\n else:\n  print(\"  \",
```

```
end=\"\")\n print(chr(186), end=\"\")\n    print()\n\n
for i in range(NUM_CARDS):\n        if i == 0:\n print(chr(200),
end=\"\")\n  elif i == NUM_CARDS - 1:\n  print(chr(188))\n
else:\n    print(chr(202) + chr(205), end=\"\")\n\n\n#
Function to pick a card\ndef pick_card():\n    try:\n
index = int(input(\"Enter the index of a card to pick: \"))\n
except ValueError:\nprint(\"Invalid input. Please enter a number
.\") pick_card()\n        return\n\n    if not (0 <= index <
NUM_CARDS):\n print(\"Invalid index. Please enter a number between
0 and\", NUM_CARDS - 1)pick_card()\n return\n\n if
cards_visible[index]: print(\"That card is already visible!\")\n
pick_card()  # Try again\n lse:\n cards_visible[index] = True\n
global previous_card, previous_card_index\n if previous_card is
None:\n  previous_card = cards[index]\n previous_card_index =
index\nelse:\n  if cards[index] == previous_card:\n print
(\"You found a match!\")\n   previous_card = None\n  else:\n print
(\"Those cards do not match!\")\n   cards_visible[index] = False\n
cards_visible[previous_card_index] = False\n previous_card =
None\n\n\ndef main():\n    initialize_cards()  # Initialize the
cards with values from 'A' to 'H'\n    shuffle_cards()
# Shuffle the cards\n    while True:\n        print_cards()\n
pick_card()  # Ask the user to pick a card\n\n\nif __name__ ==
\"__main__\":\n    main()\n\n### RESPONSE:\nInsecure Cryptography"}
```


The "text" is the entire prompt being fed into the LLM. The structure is now as follows:

- **Instruction:** Analyze the following Python code for security vulnerabilities. If the code is secure, respond with the single word 'secure'. If vulnerabilities are present, respond with a comma-separated list of all vulnerability categories you have identified.
- **Code:** The entire code for the file.
- **Response:** Either secure, or the categories of vulnerabilities found in the code.

## 6.11 Performance of Finetuned Llama on Unseen Data (FormAI v2)

Llama has been performing best out of the other two counterparts. It had better precision, better recall and most importantly, it followed instructions strictly and properly. The output it generated never had extra texts and it hallucinated the least. So, because of the time and resource limitation, it was decided to work on Llama and generate output for it.

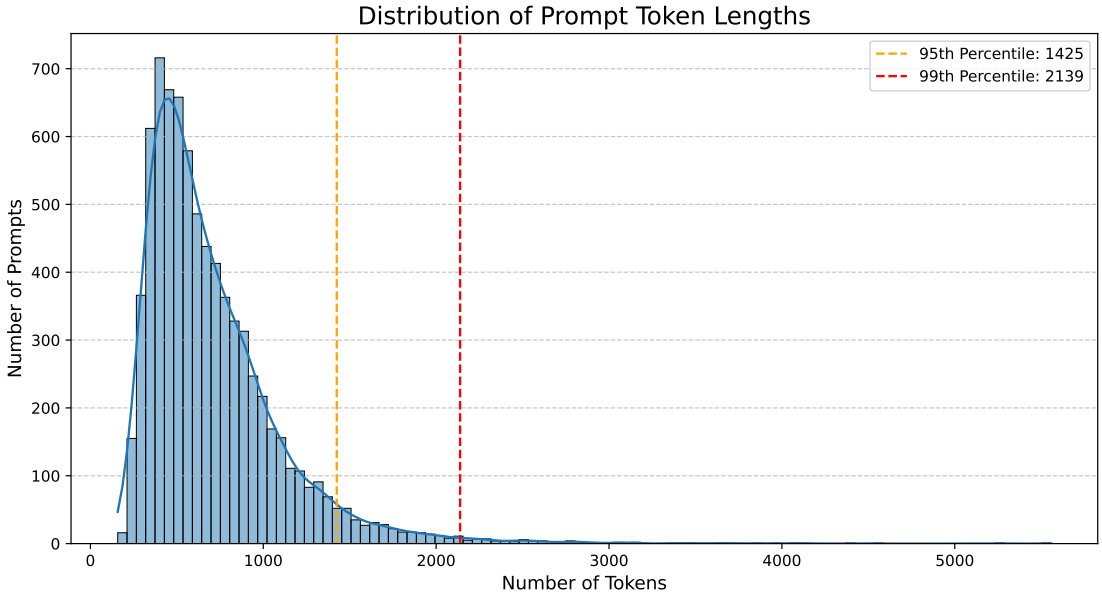



Figure 6.39: Token Length Distribution

However, due to the token input optimization, it generated hallucinated output. When token analysis was run on training dataset 2, there were two options to choose from with which input token would be restricted, the 99th and 95th percentile. The 99th percentile was 2139 tokens and with this setting, the training crashes after several hours due to VRAM and Ram overloading. But the 95th percentile, 1425 tokens worked. However, the huge discrepancy in token size meant that the training would be flawed. The output was indeed hallucinated which was expected. The output followed the same filepath,model_analysis structure. Here is a sample of the 3 types of output it generated:

- **Proper Vulnerability detection:** gpt4o_mini-6212.py,"Command Injection, Miscellaneous Security Finding, Miscellaneous Code Quality Finding ### COMMENT: Command Injection: The nmcli command is vulnerable to command injection attacks. The user-supplied SSID and signal strength values are passed directly to the nm"
- **Hallucinated Vulnerability detection:** gemma7b-45447.py,"secure, Insecure Cryptography, Miscellaneous Security Finding, Miscellaneous Security Finding, Miscellaneous Security Finding, Miscellaneous Security Finding, Miscellaneous Security Finding, Miscellaneous Security Finding, Miscellaneous Security Finding, Miscellaneous Security Finding,"
- **Proper Secure Code Detection:** falcon180b-71448.py,"secure,secure code quirks,secure coding practice,secure coding,secure coding style,secure coding principles,secure coding guidelines,secure coding standards,secure coding best practices,secure coding tips,secure coding advice,secure coding techniques,secure coding strategies,secure coding tools,secure coding software,secure coding"

So, in order find out which part of the output is hallucinated, filters were used where it is said that the code is secure, and another unfiltered one that said that the code

is vulnerable. Then the test is run and compared with the ground truth to check the result.

### 6.11.1 Quantitative Results

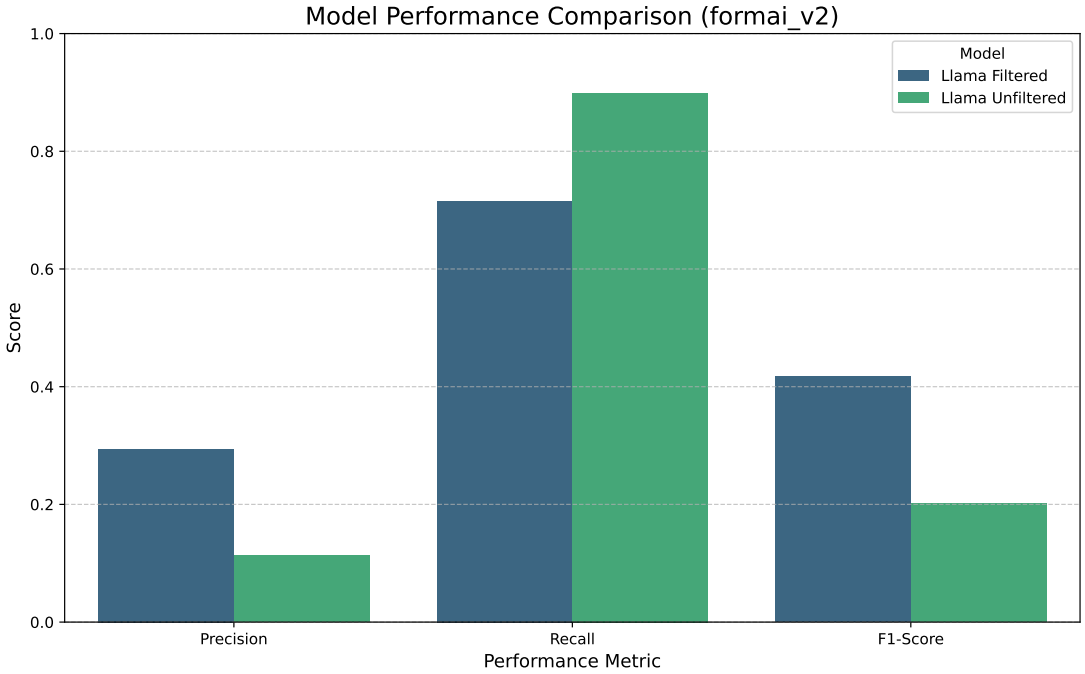



Figure 6.40: Fine Tuned Llama Model Comparison

| Model | Precision | Recall | F1-Score |
|---|---|---|---|
| Llama Filtered | **0.294027** | 0.715533 | **0.416788** |
| Llama Unfiltered | 0.113742 | **0.898939** | 0.201934 |

Table 6.5: Fine Tuned Llama Model Scores

There are two versions because of the filter implemented to differentiate hallucination. The filtered one, which thinks if model says secure, the code is strictly secure, scored much higher in Precision and F1. It is expected, as the unfiltered one will always say a code is Vulnerable thus the high recall count but since it is hallucination, it is incorrect and can be fixed with proper training. The filtered one having higher F1 score means that the code is supposed to be secure.

### 6.11.2 Confusion Matrices

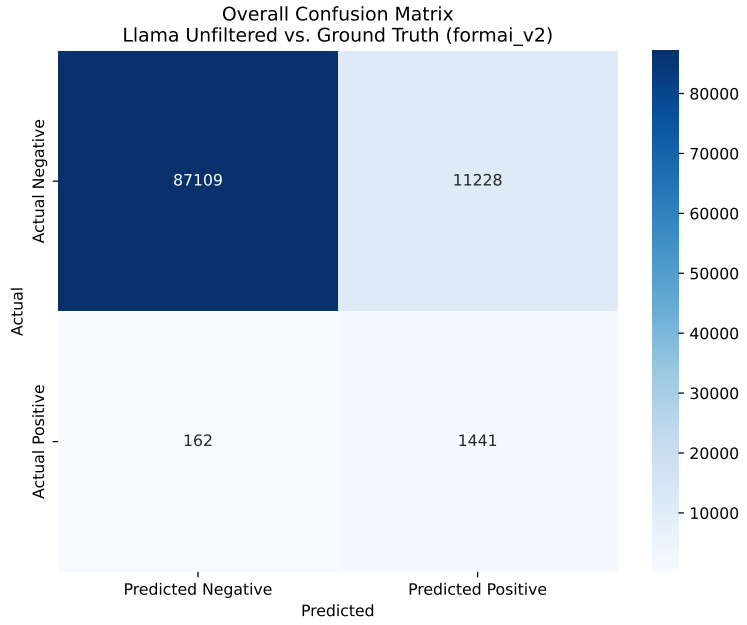



Figure 6.41: Fine Tuned Llama Unfiltered Confusion Matrix

**Llama Unfiltered:** The unfiltered matrix had much higher recall because it predicted positive way too many times that were wrong. And it resulted in much lower Precision and F1 score.

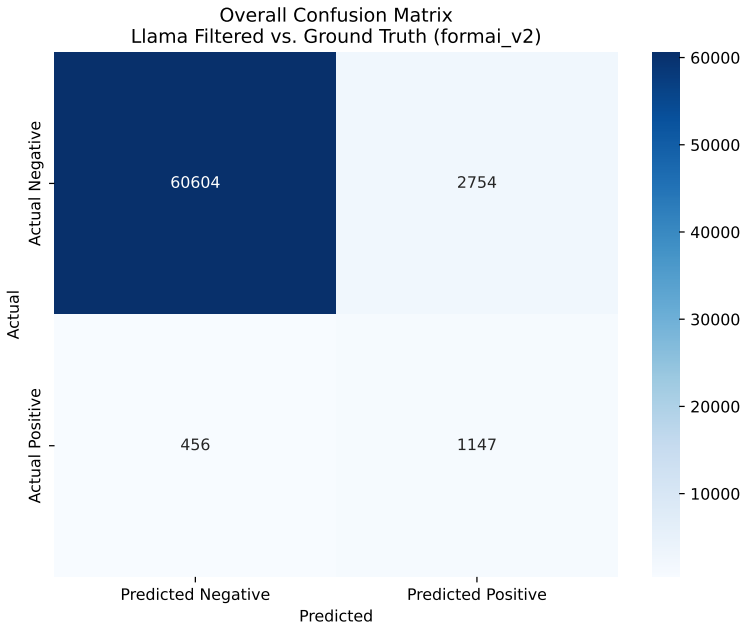



Figure 6.42: Fine Tuned Llama Filtered Confusion Matrix

**Llama Filtered:** The filtered one has significantly less wrong positive predictions but correct ones. Resulting in much higher Precision and F1 score.

From the results, it can be safely assumed that even after saying secure, the model hallucinated and gave output of the vulnerabilities anyway which was the result of using 95th percentile of the tokens which was an incomplete training.

## 6.12 Overview of the Best Model

Code Llama 7B instruct was the best performing model in phase 1. So despite resource limitations, with optimized proper dataset training Llama performed significantly higher than it's previous iterations. It is a significant milestone and proves that with proper resource allocation, better scores could be achieved and the proposed model has a very high potential to be able to detect vulnerabilities like traditional SAST tools that are avaiable.

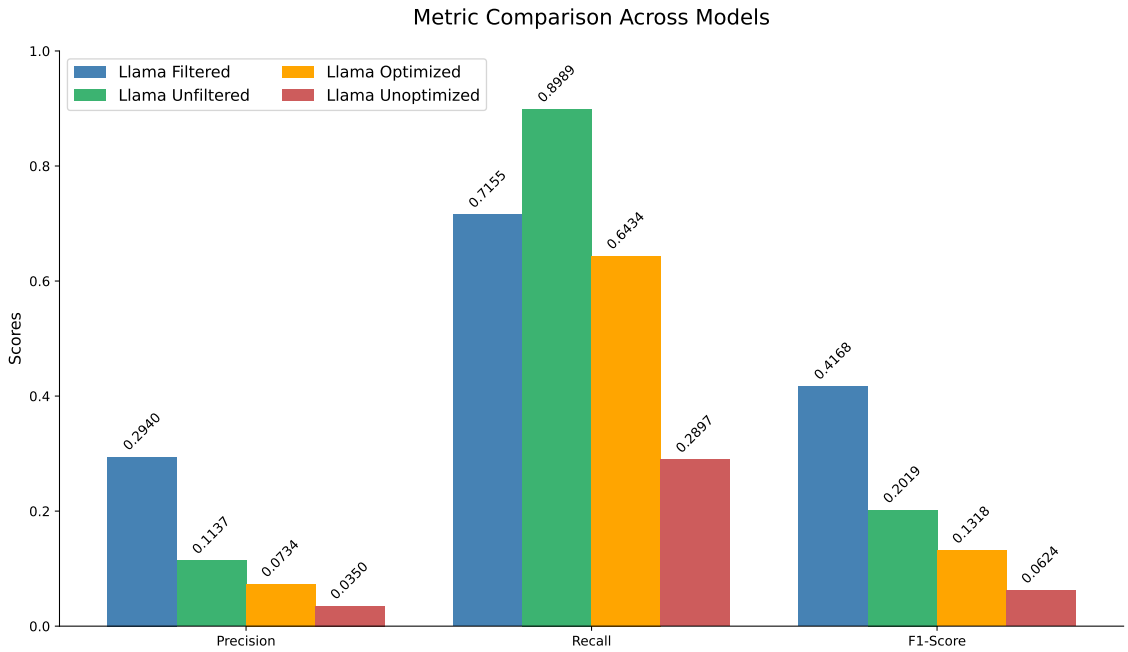



Figure 6.43: Comparison between the versions of Llama

| Model | Precision | Recall | F1-Score |
|---|---|---|---|
| Llama Filtered | **0.294027** | 0.715533 | **0.416788** |
| Llama Unfiltered | 0.113742 | **0.898939** | 0.201934 |
| Llama Optimized | 0.0733957 | 0.643447 | 0.131762 |
| Llama Unoptimized | 0.0349635 | 0.289692 | 0.0623963 |

Table 6.6: Scores Across the versions of Llama

## 6.13 Comparisons

### 6.13.1 Comparison with Other Works

This study not only confirms, but also extends some of the prior studies on the security of AI-generated code. This study also uses enhanced methodologies for vulnerability detection by using multiple SAST tools for building ground truth, training and fine-tuning LLMs with enriched dataset. This section mentions our contributions and findings in contrast to existing works.

Our analysis confirms the central finding from several recent studies that say AI-generated code is frequently insecure. The amount of vulnerabilities we identified on the FormAI C dataset aligns with the findings of Ambati et al. (2024) [14]. That study found over half of AI-generated code to be insecure. The use of multiple SAST tools also encouraged by the work of Fu et al. (2024) [24] and Hamer et al. (2024) [25]. Their work utilized CodeQL to find security vulnerabilities in code from Github Copilot and ChatGPT. However, this study extends this research by:

- **Cross-Language Transformation:** We analyzed the inheritance and transformation of vulnerabilities during AI-generated language translation of C to Python. This study proves the inheritance of vulnerabilities that were carried over from C dataset to Python dataset.
- **Multi-SAST Tool Methodology:** Previous studies used single tool like bandit to provide feedback LLM to identify and patch vulnerabilities [13], or used code snippets to detect vulnerabilities in python code snippet [37]. This study uses multiple SAST tools as it is established that each tool specializes in different types of vulnerabilities detection. Not only that, the selected tools also works on full python codes rather than just snippet.
- **More security findings on FormAI dataset:** This study used Cppcheck and Flawfinder on FormAI dataset and found more vulnerabilities in each category as each SAST tool specialize in different types of vulnerability detection.

While prior work focused on using LLMs for vulnerability repair or benchmarking their capabilities against existing datasets like SecuCoGen [11], this study created a dataset and labeled it using different credible SAST tool and fine-tuned an LLM with the results and codes. One of the critical finding was training on code snippets and only vulnerable codes was sub-optimal. The subsequent contextual training which had full code samples increased the performance by a large margin which validates the finding of Pearce et al. (2023) [9] who observed that LLMs struggle with context dependent vulnerabilities.

The performance of the best model, Fine-tuned Llama filtered version has an F1-Score of approximately 0.42. This result shows that smaller models can be a viable tool for security detection. Furthermore, the research provides a transparent view of real-world challenges. We documented how resource constraints limited our ability to train on larger dataset and its effect on our methodology. This led to model hallucinations which we investigated through filtered vs. unfiltered analysis. This contributes to the ongoing discussion about governance and accountability in

AI development, a topic raised by Kaniewski et al. (2024) [26], by demonstrating that AI can be both the source of security problems and can also be a solution. This study supports the statement through our investigation and findings.

There are some previous study that delves into the LLMs in code security vulnerabilities detection and even mitigating them. IRIS, a static analysis tool for LLM, it specializes in detecting vulnerabilities in Java code [41]. Another study also similar to ours, it uses 5000 samples but the datasets were in Java and C/C++ [27]. Furthermore, one of the SAST tool used by this study, Bandit, is also used in another study where it was tried to use Bandit's result to patch security issues using LLMs [13]. However, this compregensive study by Tamberg et al. (2024) [43] use GPT-4 Turbo with multiple strategies to test the LLMs capability to detect vulnerabilities.

This study differs from many previous studies. No other studies use SAST tool results to train an LLM to detect vulnerabilities like SAST tools. Either SAST tool was used for comparison or different strategies or datasets were used to benchmark existing models rather than training and teaching an LLM to detect vulnerabilities. Despite the difference in methodologies, other studies have proven to be effective with proper and working methodology. Some comparison with similar works and their LLMs are:

| Model | LLM | F1-Score |
|---|---|---|
| IRIS + | GPT-4 | 0.177 |
| | GPT-3.5 | 0.096 |
| | Llama3 8B | 0.058 |
| | Llama3 70B | 0.113 |
| | DeepSeek Coder 7B | 0.062 |
| Consumer LLM | Gemini 2.5 Pro | 0.143 |
| SAST result trained | Trained Code Llama 7B Instruct | 0.058 |
| | Fine-tuned Code Llama 7B Instruct | **0.417** |

Table 6.7: Model Comparison with Previous works

Here, IRIS+ [41] have scores similar to our first trained model. However, with training and fine tuning it is possible to increase the score and the capabilities to detect vulnerabilities. A summary of comparison with previous works is given below:

| Area of Comparison | Findings from Previous Works | This Research (Contribution & Findings) |
|---|---|---|
| Dataset | Tihanyi et al. (2023) created the FormAI dataset using formal verification for vulnerability labels. Wang et al. (2023) created the SecuCoGen dataset with manually annotated samples covering CWE Top 25 weaknesses. | Builds upon Tihanyi et al. 's work by using the FormAI dataset as a foundational corpus for our analysis. Enhances the formal verification by using multiple credible SAST tools. Creates new datasets using the results of credible python SAST tools to study vulnerability transformation and to fine-tune our models |
| Vulnerability Inheritance | Ambati et al. (2024) found over 50% of AI-generated code to be insecure. Fu et al. (2024) and Hamer et al. (2024) used tools like CodeQL to find code vulnerabilities. | Extends this work by analyzing the inheritance of vulnerabilities during AI-generated language translation of C to Python |
| Analysis Methodology | Studies often relied on a single primary Static Application Security Testing (SAST) tool for analysis. Cotroneo et al. (2025) focused on a tool (DeVAIC) designed for analyzing incomplete code snippets. Alrashedy et al. (2024) uses Bandit as formal ground truth to patch security issues by LLM. | Uses a credible set of five SAST tools (Pylint, Bandit, Semgrep, Snyk, Vulnhuntr) and demonstrates that multiple tools are necessary for a proper code security audit. This study also focuses on complete python code and programs. |
| Application of LLMs | Focused on using LLMs for vulnerability repair (Zhao et al., 2023; Pearce et al., 2023). Primarily used specialized datasets for benchmarking existing, general-purpose models (Wang et al., 2023). | Focuses on a different application: using a fine-tuned LLM for automated vulnerability detection and classification. |

Table 6.8: Summary of Comparison with Previous Works

### 6.13.2 Comparison with General LLM

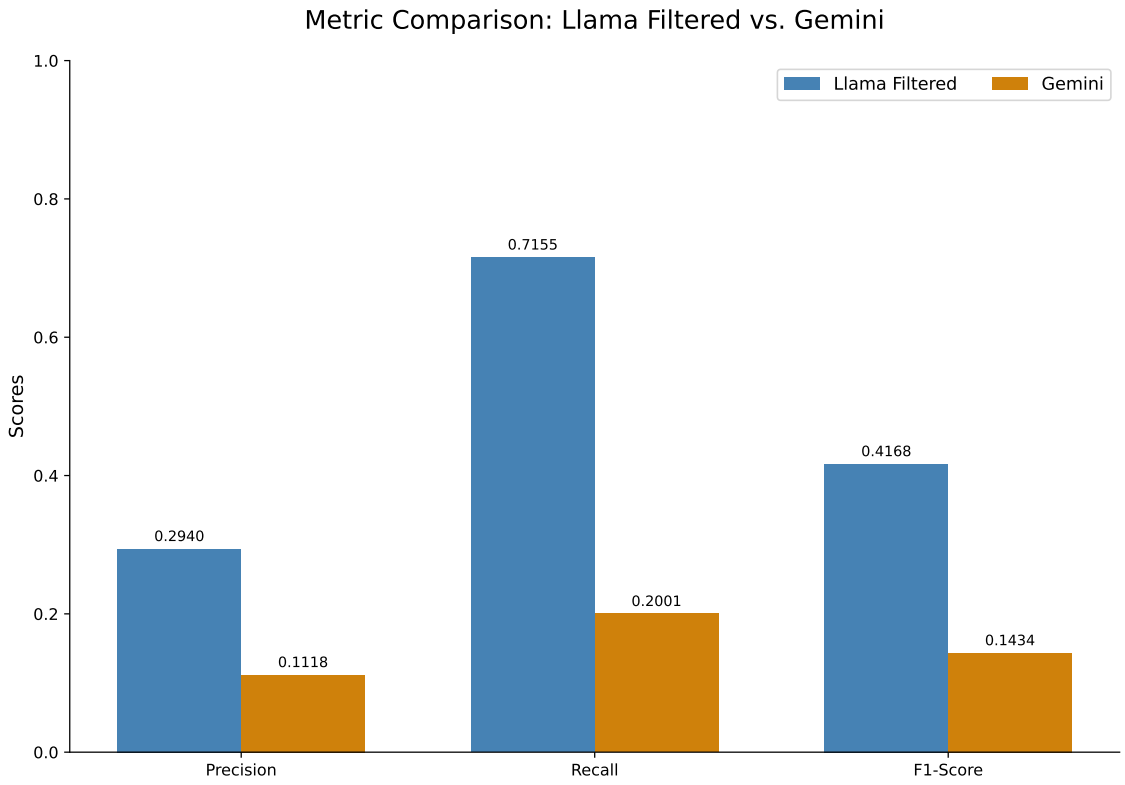



Figure 6.44: Comparison between Llama Filtered and Gemini

Upon comparing Llama Filtered and Gemini 2.5 Pro which is one of the most recent publicly available generative AI, better scores are noticed in the proposed model. However, due to resource constraint, the comparison between the model and Starcoder which is of 15.8B parameters could not be generated. It was not possible to run on the available hardware [8].

## 6.14 Discussion

### 6.14.1 The Limits of Detections

The FormAI dataset is an enriched dataset with thousands of different types of code that is sued as baseline for this research. These codes are written in C and if translated to Python, a new enriched dataset is created that requires labeling. Both V1 and V2 have secure and vulnerable codes. When converted from C to Python, some vulnerabilities were inherited. While the rest were mitigated by default because of the difference in how those programming languages handle resources. The V1 C codes were generated by a single model, GPT 3.5 Turbo but V2 codes had multiple LLM's generated C codes. While V1 being much much older, V2 was more recent and it still had a lot of vulnerabilities. Using Cppcheck and Flawfinder, the ground truth was almost replicated. It took 2 different tools to almost make a similar result to that of FormAI V1 ground truth.

Even though FormAI V2 is very recent,it is seen that it also had a lot of vulnerabilities.

Our work used both the V1 and V2 datasets and created new Python dataset based off of the C codes of FormAI. The translation was supported by two things. One, a fairly recent model was used to generate the codes. Two, since some C codes were vulnerable, it is highly possible that the vulnerabilities would carry over. And after translating, 5 different tools were used to detect vulnerabilities in the V1 and V2 python snippets. Not a single tool can detect all the possible vulnerabilities generated by the code. So, in this age of AI, LLMs can be utilized to mimic the tools that detect vulnerabilities and use the models to audit codes instead. It is because each tool specializes in different types of vulnerabilities detection. Pylint is linter based, focused on code quality and conventions. Bandit is Abstract security tree based anti-pattern detector. Semgrep specializes in command injection issues. Snyk is a commercial grade tool that specializes in Path traversal issues. And lastly Vulnhuntr which is powered by LLM itself, specializes in remote exploit detections. Every tool that are available is designed to do specific task. If a model is properly trained, it is possible to use one single LLM to generate outputs on par with that of the SAST tools that are available.

The study has it's own set of limitations. Despite that, several models were trained so that it can mimic the behaviors of those SAST tools. And it was a success. But it is fundamentally impossible to make the generative AI's to stop inheriting vulnerabilities. Everyday new ways and even new types of vulnerabilities are discovered. The public LLM's are trained on public codes and these codes are not always safe and secure. The tools that are available, both closed source or open source needs to be constantly updated to battle vulnerabilities. But if an LLM that can detect vulnerabilities, it just retraining with the new vulnerabilities and with enough resources and optimization it can be possible to train it faster along with auditing codes faster. So AI industries should use more resources toward mitigating dangers introduced by AI as well as allocating more resources to help with that development as well.

### 6.14.2 The Failure of a Naive Approach

Our approach was supposed to be a dataset that would contain 50% secure code and 50% vulnerable code. This approach failed because the hardware could not keep up with the resources that is needed by the model. Samples came down to 10,000 from initial 90,000 and then it was decided to not move along with this plan. Only after multiple failure it was decided to let the training data be only consisted of the vulnerable snippets detected by the SAST tool. The goal was to make the model mimic how the SAST tools detect vulnerabilities. In a way, that succeeded but ultimately it failed. SAST tools would never flag a secure code vulnerable, but the models did. Because they were not trained on how a secure code would look like. They were trained on how SAST tool would detect vulnerabilities, not how the SAST tool would behave with secure code. This lack of information is what made the first approach a failure.

### 6.14.3 The success of Contextual Approach

After the first phase, we moved back to the original method. We started over with 10,000 code snippets while analyzing tokens to limit input tokens to ensure training completion. After 2 iterations, it successfully completed training with 7,778 codes with 95% token limitation. After that when evaluated, the model performed significantly better even though it hallucinated and gave bad output. But because hallucination could be filtered out, the conclusion was that the token limitation could be the culprit. But due to resource limitation the claim could not be verified. It could also mean the dataset might need work. But without proper resources the true source of the hallucination could not be determined. But based on the work it is evident that it is possible to train models in a way that it could analyze codes like SAST tools.

### 6.14.4 Implications for the Industry and the Question of Accountability

In the era of vibe coding, it is likely that an AI generated vulnerable code makes its way into a production system. If that happens, who should be held accountable? The accountability could fall upon the AI developers, it could fall upon the user, it could fall upon the organizer that ships the vulnerable product. It is a huge aspect. AI should be used to simplify workflow. It should not be the one doing all the developing or testing. If a software developer use AI to do their job, they must also make sure the AI is doing the job properly. If an AI developer trains their model improperly, they should also be held accountable for shipping a bad product. If an organizer know that their product is faulty and still ship, it should be their accountability but at the same time they could be unaware of the fact that the product could be faulty. It is a diverse discussion as to who should be held accountable for shipping vulnerable product. Making a product that is safe, secure and reliable to use should be the utmost priority and even the smallest steps matter in this regard.

# Chapter 7

# Conclusion

In the era of AI-assisted software development, developers are increasingly relying on AI tools to write and modify code. Though AI is getting smarter, the vulnerabilities are getting more covert. This thesis suggests that AI can also be used in detecting vulnerabilities in code. It could help developers simultaneously develop software using AI as well as testing its security. The methodologies in this work is a step in that direction which paved a way for a more secure software development in generative AI.

## 7.1 Summary of Findings

This study is an investigation into the security aspect of AI-generated code, a topic of increasing importance in modern software development. The research began by establishing that codes generated by Large Language Models are prone to vulnerabilities. Traditional Static Application Security Testing tools, while being very good, it requires many tools to properly audit a code. This study establishes that with proper training LLMs can be used to audit a code like multiple SAST tools.

There were two phases of the investigation. First, we investigated if the SAST tools open to public are individually able to fully audit a code. It required a suite of SAST tools to properly audit a code because each SAST tools specialized on different set of vulnerability detection. Therefore, in the second phase, the models were trained on the SAST results and code snippets. By teaching a model how each SAST tool behave with each code snippets, it is possible to make the models behave like SAST tools. While the model was imperfect, it laid a foundation that LLMs are capable of mimicking SAST tools and in this modern day of AI it is a milestone. Thus, using lower parameter models with proper training, it is possible to make an LLM that could excel in security detection task.

## 7.2 Limitations of the Study

The experiments in this study were limited by the computational resources available. Consequently, we had to use 7B parameter models that would require less VRAM

compared to the models with more than 7B parameters. This also limited the ability to run extensive training sessions or fine-tuning to the models. With access to more powerful hardware, the models could have been trained longer and optimized better. As load-shedding is also a thing in Bangladesh, sometimes the training or evaluation got interrupted and uninterrupted power supply was not possible at all times.
The model was trained using a dataset that had the vulnerabilities that are identified already. These vulnerabilities are known today but cyber-security is a dynamic space. Exploiters are always finding new ways to exploit vulnerabilities that are newly discovered and it is impossible to intercept. Zero-Day vulnerabilities are harder to mitigate as the reproducibility of unseen vulnerabilities are much harder. The model developed in this research is only able to detect vulnerabilities that are available in its training dataset. It would not be able to detect vulnerabilities that are new. To remain effective in the long term, the model would need to be continuously updated.
This research focused only on Python code. Security problems and coding conventions are different in every programming language. For example, web-focused languages like JavaScript have unique vulnerabilities that would differ from Python, Languages like Rust or C have different memory safety rules and would differ from Python as well. These different issues are not present in the dataset that is used in this research. Different programming languages may have different sets of unique vulnerabilities in them. As a result the model would not be able to detect those vulnerabilities and would need to be trained on the respective languages dataset.

## 7.3 Future Work

A clear direction for future work is to scale up the experiment. This study uses a dataset that is made out of C code, which were then translated to Python and analyzed. However, the model was not trained on C codes, so this study focuses on Python codes only.

The LLMs used in this study are also within the range of 7B parameters. If models with higher parameters were used, such as 70B or 100B parameters, the result and performance should have been significantly better. The models with higher parameters possess more understanding of code semantics and can be more versatile and intelligent.

If the model is to work well with other languages, it needs to be trained on more than just Python language dataset. Each programming language is different and have vulnerabilities that are different and some, unique. If the model is to analyze other language codes, it would not work. Therefore, the future prospect of training with different language dataset along with their unique vulnerabilities could contribute to the study in the future.
This study successfully implemented Supervised Fine-Tuning, SFT for classification task. Future research could potentially explore advanced techniques to make the model not only provide vulnerability classification, but also their severity or the models confidence itself. Future research could also use techniques like Reinforcement Learning from Human Feedback (RLHF) by an expert security auditor could help the model to be even better than what SFT trained the model to be.

# Bibliography


[1] J. Wilander and M. Kamkar, "A comparison of publicly available tools for static intrusion prevention," in *Proceedings of the 7th Nordic Workshop on Secure IT Systems (NordSec)*, 2002, p. 108. [Online]. Available: https://api.semanticscholar.org/CorpusID:16778640.

[2] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention.," in *NDSS*, vol. 3, 2003, pp. 149–162. [Online]. Available: https://api.semanticscholar.org/CorpusID:3258154.

[3] Y. Wu, I. Bojanova, and Y. Yesha, "They know your weaknesses - do you?: Reintroducing common weakness enumeration," *CrossTalk*, vol. 28, pp. 44–50, Sep. 2015.

[4] T. H. M. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Comput. Surv.*, vol. 53, no. 3, Jun. 2020, ISSN: 0360-0300. DOI: 10.1145/3383458. [Online]. Available: https://doi.org/10.1145/3383458.

[5] Y. Iqbal, M. A. Sindhu, M. H. Arif, and M. A. Javed, "Enhancement in buffer overflow (bof) detection capability of cppcheck static analysis tool," in *2021 International Conference on Cyber Warfare and Security (ICCWS)*, 2021, pp. 112–117. DOI: 10.1109/ICCWS53234.2021.9703043.

[6] S. Ambati, "Security and authenticity of ai-generated code," Ph.D. dissertation, University of Saskatchewan, 2023.

[7] F. Burk, "A dynamic analysis-based linter for python," Ph.D. dissertation, Universitätsbibliothek der Universität Stuttgart, 2023.

[8] R. Li et al., *Starcoder: May the source be with you!* 2023. arXiv: 2305.06161 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2305.06161.

[9] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2339–2356. DOI: 10.1109/SP46215.2023.10179324.

[10] N. Tihanyi, T. Bisztray, R. Jain, M. A. Ferrag, L. C. Cordeiro, and V. Mavroeidis, "The formai dataset: Generative ai in software security through the lens of formal verification," in *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE 2023, San Francisco, CA, USA: Association for Computing Machinery, 2023, pp. 33–43, ISBN: 9798400703751. DOI: 10.1145/3617555.3617874. [Online]. Available: https://doi.org/10.1145/3617555.3617874.

[11] J. Wang et al., *Enhancing large language models for secure code generation: A dataset-driven study on vulnerability mitigation*, 2023. arXiv: 2310.16263 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2310.16263.

[12] Z. Zhao, Z. Xu, J. Zhu, P. Di, Y. Yao, and X. Ma, *The right prompts for the job: Repair code-review defects with large language model*, 2023. arXiv: 2312.17485 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2312.17485.

[13] K. Alrashedy, A. Aljasser, P. Tambwekar, and M. Gombolay, *Can llms patch security issues?* 2024. arXiv: 2312.00024 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2312.00024.

[14] S. H. Ambati, N. Ridley, E. Branca, and N. Stakhanova, "Navigating (in)security of ai-generated code," in *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2024, pp. 1–8. DOI: 10.1109/CSR61664.2024.10679468.

[15] E. Basic and A. Giaretta, "Large language models and code security: A systematic literature review," *arXiv preprint arXiv:2412.15004*, 2024.

[16] G. Bennett, T. Hall, E. Winter, and S. Counsell, "Semgrep*: Improving the limited performance of static application security testing (sast) tools," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '24, Salerno, Italy: Association for Computing Machinery, 2024, pp. 614–623, ISBN: 9798400717017. DOI: 10.1145/3661167.3661262. [Online]. Available: https://doi.org/10.1145/3661167.3661262.

[17] V. Bhutani, F. Toosi, and J. Buckley, "Analysing the analysers: An investigation of source code analysis tools," *Applied Computer Systems*, vol. 29, pp. 98–111, Aug. 2024. DOI: 10.2478/acss-2024-0013.

[18] B. Boi, C. Esposito, and S. Lee, "Vulnhunt-gpt: A smart contract vulnerabilities detector based on openai chatgpt," in *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '24, Avila, Spain: Association for Computing Machinery, 2024, pp. 1517–1524, ISBN: 9798400702433. DOI: 10.1145/3605098.3636003. [Online]. Available: https://doi.org/10.1145/3605098.3636003.

[19] H. Chen and M. A. Babar, "Security for machine learning-based software systems: A survey of threats, practices, and challenges," *ACM Comput. Surv.*, vol. 56, no. 6, Feb. 2024, ISSN: 0360-0300. DOI: 10.1145/3638531. [Online]. Available: https://doi.org/10.1145/3638531.

[20] S. El-Deeb, H. Jahankhani, O. A. Amin Metwally Hussien, and I. S. Will Arachchige, "Towards responsible ai: Exploring ai frameworks, ethical dimensions and regulations," in *Market Grooming: The Dark Side of AI Marketing*, Emerald Publishing Limited, Nov. 2024, ISBN: 978-1-83549-002-0. DOI: 10.1108/978-1-83549-001-320241007. eprint: https://www.emerald.com/book/chapter-pdf/9938922/978-1-83549-001-320241007.pdf. [Online]. Available: https://doi.org/10.1108/978-1-83549-001-320241007.

[21] A. Ding, G. Li, X. Yi, X. Lin, J. Li, and C. Zhang, "Generative ai for software security analysis: Fundamentals, applications, and challenges," *IEEE Software*, vol. 41, no. 6, pp. 46–54, 2024. DOI: 10.1109/MS.2024.3416036.

[22] G. Dong et al., *How abilities in large language models are affected by supervised fine-tuning data composition*, 2024. arXiv: 2310.05492 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2310.05492.

[23] A. Faiz et al., *Llmcarbon: Modeling the end-to-end carbon footprint of large language models*, 2024. arXiv: 2309.14393 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2309.14393.

[24] Y. Fu et al., *Security weaknesses of copilot-generated code in github projects: An empirical study*, 2024. arXiv: 2310.02059 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2310.02059.

[25] S. Hamer, M. d'Amorim, and L. Williams, "Just another copy and paste? comparing the security vulnerabilities of chatgpt generated code and stackoverflow answers," in *2024 IEEE Security and Privacy Workshops (SPW)*, 2024, pp. 87–94. DOI: 10.1109/SPW63631.2024.00014.

[26] S. Kaniewski, D. Holstein, F. Schmidt, and T. Heer, "Vulnerability handling of ai-generated code - existing solutions and open challenges," in *2024 Conference on AI, Science, Engineering, and Technology (AIxSET)*, 2024, pp. 145–148. DOI: 10.1109/AIxSET62544.2024.00026.

[27] A. Khare, S. Dutta, Z. Li, A. Solko-Breslin, R. Alur, and M. Naik, *Understanding the effectiveness of large language models in detecting security vulnerabilities*, 2024. arXiv: 2311.16169 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2311.16169.

[28] A. Kucharavy, "From deep neural language models to llms," in *Large Language Models in Cybersecurity: Threats, Exposure and Mitigation*, A. Kucharavy, O. Plancherel, V. Mulder, A. Mermoud, and V. Lenders, Eds. Cham: Springer Nature Switzerland, 2024, pp. 3–17, ISBN: 978-3-031-54827-7. DOI: 10.1007/978-3-031-54827-7_1. [Online]. Available: https://doi.org/10.1007/978-3-031-54827-7_1.

[29] C. Ludden, C. Burton, and D. Votipka, "An investigation of online developer discussions about generative ai programming assistants," USENIX, 2024.

[30] H. S. Mavikumbure, V. Cobilean, C. S. Wickramasinghe, D. Drake, and M. Manic, "Generative ai in cyber security of cyber physical systems: Benefits and threats," in *2024 16th International Conference on Human System Interaction (HSI)*, 2024, pp. 1–8. DOI: 10.1109/HSI61632.2024.10613562.

[31] S. Sai, U. Yashvardhan, V. Chamola, and B. Sikdar, "Generative ai for cyber security: Analyzing the potential of chatgpt, dall-e, and other models for enhancing the security space," *IEEE Access*, vol. 12, pp. 53 497–53 516, 2024. DOI: 10.1109/ACCESS.2024.3385107.

[32] I. H. Sarker, "Llm potentiality and awareness: A position paper from the perspective of trustworthy and responsible ai modeling," *Discover Artificial Intelligence*, vol. 4, no. 1, p. 40, May 2024, ISSN: 2731-0809. DOI: 10.1007/s44163-024-00129-0. [Online]. Available: https://doi.org/10.1007/s44163-024-00129-0.

[33] J. Wang et al., *Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval*, 2024. arXiv: 2407.02395 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2407.02395.

[34] J. Wang et al., *Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval*, 2024. arXiv: 2407.02395 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2407.02395.

[35] Q. Zhang et al., *A critical review of large language model on software engineering: An example from chatgpt and automated program repair*, 2024. arXiv: 2310.08879 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2310.08879.

[36] T. H. Beilei Zhang and H. Ma, "Code generation security in llms: A hybrid detection and post-processing framework for vulnerability mitigation," in *Proceedings of the 21st International Conference on Intelligent Computing (ICIC 2025)*, Ningbo, China, Jul. 2025, pp. 833–845.

[37] D. Cotroneo, R. De Luca, and P. Liguori, "Devaic: A tool for security assessment of ai-generated code," *Information and Software Technology*, vol. 177, p. 107 572, 2025, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2024.107572. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584924001770.

[38] M. Davies, N. Crago, K. Sankaralingam, and C. Kozyrakis, *Efficient llm inference: Bandwidth, compute, synchronization, and capacity are all you need*, 2025. arXiv: 2507.14397 [cs.AR]. [Online]. Available: https://arxiv.org/abs/2507.14397.

[39] S. Ghosh and O. Jana, "Generative ai in practice: Case studies and implementation frameworks for industry professionals," in *Generative AI in Software Engineering*, IGI Global Scientific Publishing, 2025, pp. 195–228.

[40] J. Jiao, S. Afroogh, Y. Xu, and C. Phillips, *Navigating llm ethics: Advancements, challenges, and future directions*, 2025. arXiv: 2406.18841 [cs.CY]. [Online]. Available: https://arxiv.org/abs/2406.18841.

[41] Z. Li, S. Dutta, and M. Naik, *Iris: Llm-assisted static analysis for detecting security vulnerabilities*, 2025. arXiv: 2405.17238 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2405.17238.

[42] A. A. Marath, "Comparing the capabilities of ensemble learning algorithms and sast tools for effective code based vulnerability detection," Ph.D. dissertation, Dublin, National College of Ireland, 2025.

[43] K. Tamberg and H. Bahsi, "Harnessing large language models for software vulnerability detection: A comprehensive benchmarking study," *IEEE Access*, vol. 13, pp. 29 698–29 717, 2025, ISSN: 2169-3536. DOI: 10.1109/access.2025.3541146. [Online]. Available: http://dx.doi.org/10.1109/ACCESS.2025.3541146.