Comparison of Static Application Security Testing Tools and Large Language Models for Repo-level Vulnerability Detection

Xin Zhou
Singapore Management University
Singapore
xinzhou.2020@phdcs.smu.edu.sg

Duc-Manh Tran
Hanoi University of Science and Technology
Vietnam
manh.td120901@gmail.com

Thanh Le-Cong
The University of Melbourne
Australia
congthanh.le@student.unimelb.edu.au

Ting Zhang
Singapore Management University
Singapore
tingzhang.2019@phdcs.smu.edu.sg

Ivana Clairine IRSAN
Singapore Management University
Singapore
ivanairsan@smu.edu.sg

Joshua SUMARLIN
Singapore Management University
Singapore
jsumarlin.2022@scis.smu.edu.sg

Bach Le
University of Melbourne
Australia
bach.le@unimelb.edu.au

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Manan Suri
manans@umd.edu

Software vulnerabilities in 2013 vs 2023

5,697

2013

29,065

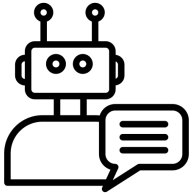
2023

With an increasing number of software vulnerabilities, timely detection is crucial for mitigating economic losses and safeguarding critical infrastructure.

Automated Software Vulnerability Detection

Static Application Security Testing (SAST) Tools

- Traditional, widely used, low-cost, ability to find bugs without running the program



Large Language Models (LLMs)

- Extensive knowledge owing to large scale pre-training, emergent code understanding abilities

However, despite considerable interest in either SAST Tools or LLMs for vulnerability detection, before this study, there was no comprehensive comparative study between the two.

Why?

1.

Lack of formulation for repo-level detection using DL-based methods (focus on individual functions).

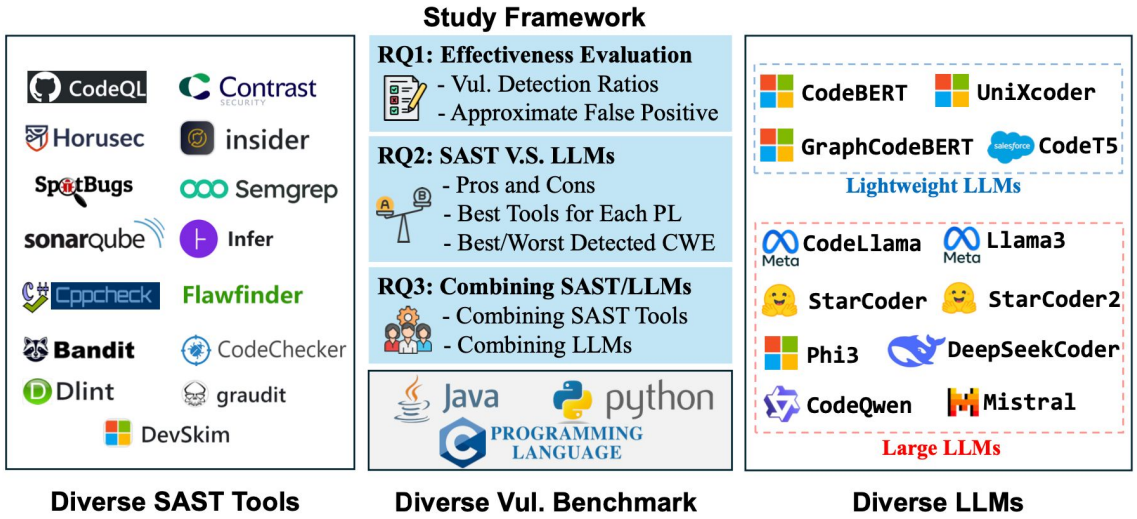
2.

Lack of datasets supporting both approaches (SAST and LLMs focus on different levels of code granularity).

3.

No unified LLM evaluation framework that covers the diversity of models and techniques

Overview of the Study



- **RQ1:** How effective are SAST tools and LLMs?
- **RQ2:** Which approach (SAST or LLM) is better for vulnerability detection across different programming languages?
- **RQ3:** Can combining SAST tools and LLMs improve detection?

"We compared 15 SAST tools and 12 LLMs for detecting vulnerabilities in Java, C, and Python repositories."

Study Design Repo-level Vulnerability Detection

- **Traditional SAST Tools:** Detect vulnerabilities across entire repositories without executing the code.
- **Current LLM Methods:** Focus on function-level vulnerability detection, targeting individual functions in isolation.
- **Their Study:** Introduces repo-level vulnerability detection for LLMs, enabling predictions across an entire repository, similar to SAST functionality.
- **Approach for LLMs:**
  - Split repositories into functions.
  - Detect vulnerabilities at the function level.
  - Aggregate predictions for comprehensive insights.

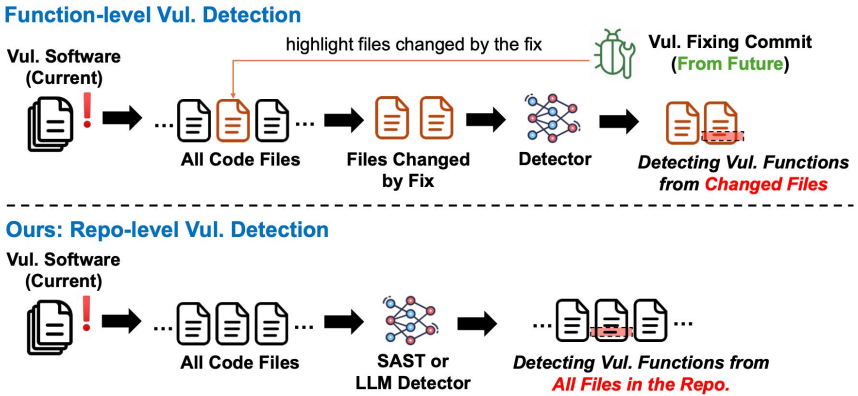


Figure 2: Repo-level and Function-level Vul. Detection

Study Design Repo-level Vulnerability Detection

Java & C Datasets:

Curated from real-world repositories with CVE IDs, derived from the works of Li et al. and Lipp et al.

Python Dataset:

A newly constructed dataset from the National Vulnerability Database (NVD), containing real-world vulnerabilities and their fixing commits.

Data Parsing Methodology:

Each dataset is parsed into function-level code snippets using Tree-sitter for accurate vulnerability labeling.

Table 1: Statistics of the Used Datasets.

Dataset	#Project	#CVE	#Functions	Vul. Ratio
Java	50	87	3,986,848	0.008%
C	6	85	1,181,556	0.009%
Python	78	98	312,045	0.18%

Study Design SAST Tools Used

Java Tools

CodeQL, Contrast Codesec, Horusec, Insider, SpotBugs, Semgrep, SonarQube

C Tools

Flawfinder, Cppcheck, Infer, CodeChecker, CodeQL

Python Tools

Bandit, Dlint, DevSkim, CodeQL, Graudit, Semgrep

Key Strengths of SAST Tools:

- Low false positives
- Fast analysis
- Widely used in real-world applications for detecting software vulnerabilities.

Study Design LLMs Used

Lightweight LLMs (<1B parameters):

CodeBERT, GraphCodeBERT, CodeT5, UniXcoder.

Large LLMs (≥1B parameters):

StarCoder, CodeLlama, Mistral, DeepSeek-Coder, Llama3, StarCoder2, CodeQwen, Phi3.

Table 2: Overview of Studied LLMs.

	Model	#Param.	Type	Release Date	Code Gene. (HumanEval) Pass@1	Adaptation Techniques
Light-weight LLMs (<1B)	CodeBERT	0.13B	Enc	Jul. 2020	-	Full Fine-tuning
	G-BERT	0.13B	Enc	Feb. 2021	-	
	CodeT5	0.22B	Enc-Dec	Sep. 2021	-	
	UniXcoder	0.13B	Enc-Dec	Mar. 2022	-	
Large LLMs (≥1B)	StarCoder	7B	Dec	May 2023	20.4%	PEFT, Zero-shot Prompt, Few-shot Prompt, CoT Prompt
	CodeLlama	7B	Dec	Aug. 2023	45.7%	
	Mistral	7B	Dec	Sep. 2023	-	
	DSCoder	6.7B	Dec	Dec. 2023	80.2%	
	StarCoder2	7B	Dec	Feb. 2024	34.1%	
	CodeQwen	7B	Dec	Apr. 2024	50.8%	
	LLama3	8B	Dec	Apr. 2024	62.2%	
	Phi3	3.8B	Dec	May 2024	58.5%	

Study Design LLM Adaptation Techniques

Prompt-Based Methods:

- Zero-Shot: Detect vulnerabilities without any task-specific data.
- Few-Shot: Provide a few labeled examples to guide detection.
- Chain-of-Thought (CoT): Guide models with step-by-step reasoning prompts.

Fine-Tuning Methods:

- Full Fine-Tuning: Applied for lightweight models (<1B parameters).
- Parameter-Efficient Fine-Tuning (LoRA): Used for large models (>1B parameters) to update only a subset of parameters, reducing computational cost.

Experimental Setup Vulnerability Detection Scenarios

The performance of SAST tools and LLMs was evaluated across two distinct detection scenarios.

- **Scenario 1 (S1):**

  A vulnerability is detected if **any** vulnerable function within the repository is identified.

- **Scenario 2 (S2):**

  A vulnerability is only detected if **all** vulnerable functions in the repository are identified.

Experimental Setup Evaluation Metrics

Following past work, they use the following metrics:

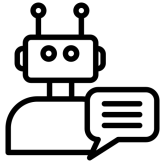
$$Vuln.\ Detection\ Ratio = \frac{\#Detected\ Vuln.}{\#All\ Vuln.\ in\ Benchmark}$$

$$Marked\ Function\ Ratio = \frac{\#Marked\ Function}{\#All\ Functions\ in\ Benchmark}$$

Results RQ1: Effectiveness of SAST Tools and LLMs

SAST Tools

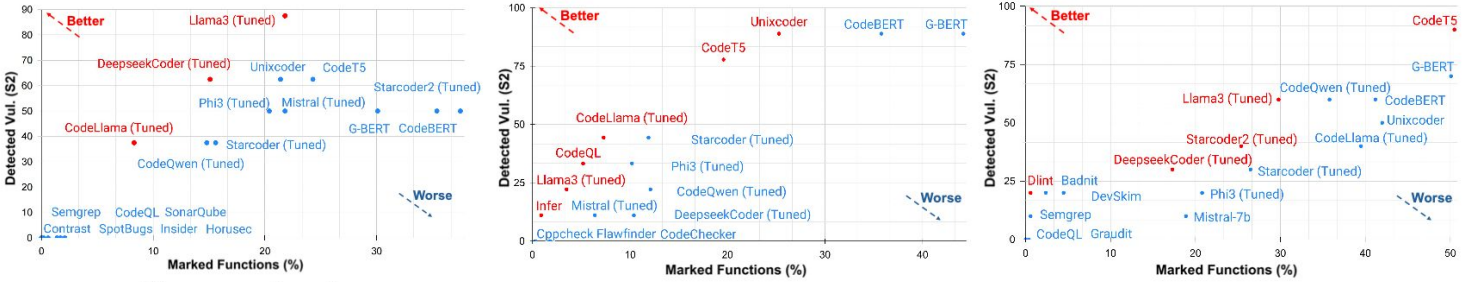
- Lower detection rates
- Reduced false positives
- Detection rates up to 44.4%
- Marked function ratio up to 5.2%



LLMs

- Detected more vulnerabilities
- Higher false positives
- Detection rates up to 100%
- Marked function ratio up to 77.4%

Results RQ2: SAST Tools vs LLMs



(a) Java Benchmark (b) C Benchmark (c) Python Benchmark

Figure 3: The Relationship between the Vulnerability Detection Ratios (Scenario 2) and the Marked Function Ratios

Best tools for each language:

Java

Finetuned DeepseekCoder

C

Finetuned UniXCoder

Python

Finetuned Llama3

Results RQ2: SAST Tools vs LLMs

- **CWE 119 (Buffer Overflow)** LLMs significantly outperform SAST tools in detecting this vulnerability across both scenarios, especially in stricter detection (Scenario 2).
- **CWE 79 (Cross-Site Scripting)** LLMs excel under less strict conditions but show superior performance when all vulnerabilities must be detected (Scenario 2).
- **CWE 476 (NULL Pointer Dereference)** Much better detection rates are observed with LLMs, particularly in Scenario 2.
- **Challenges with CWE 835 (Infinite Loop) and CWE 89 (SQL Injection)** Both LLMs and SAST tools struggle to effectively detect these vulnerabilities, indicating ongoing challenges in these classes.

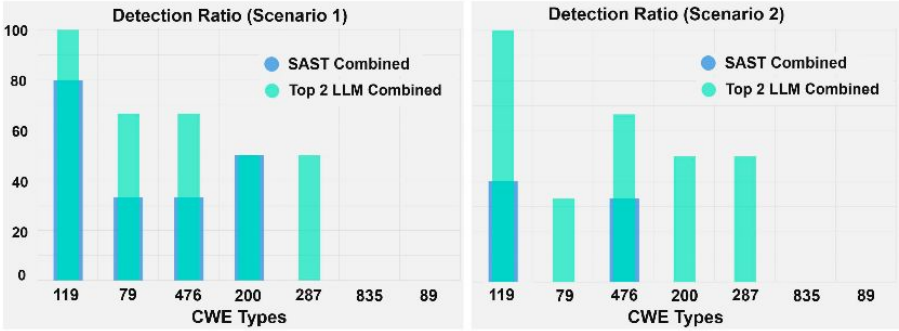


Figure 4: Vulnerability Detection Ratios in Different Classes classes. Both the SAST tool group and the 2 optimal LLMs group failed to detect any vulnerabilities in these classes.

Results RQ3: Combining SAST Tools or LLMs

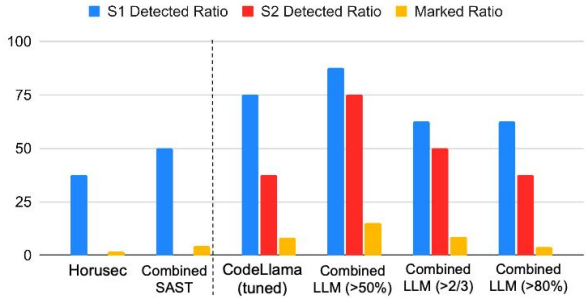


Figure 5: Combinations of SAST Tools or LLMs for Java

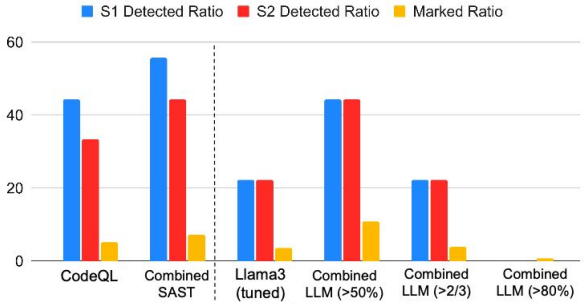


Figure 6: Combinations of SAST Tools or LLMs for C

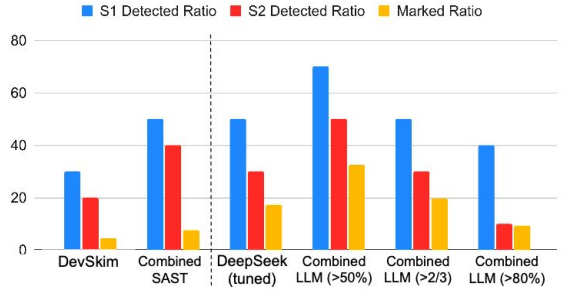


Figure 7: Combinations of SAST Tools or LLMs for Python

- Combining SAST tools boosts detection rates from 25.2% to 100.0%.
- Combining LLMs reduces the marked function ratio by 40.9% to 74.6% on average.
- LLMs are most effective for Java, while SAST combinations perform better for C and Python.

Discussion Evaluating ChatGPT

- Focus of the study was on open-source LLMs due to cost and reproducibility challenges.
- Conducted small-scale experiments with ChatGPT (gpt-3.5-turbo-0125).

Table 4: Evaluating ChatGPT on A Small Detection Range.

Pos:Neg =1:100	CodeBERT (Tuned)	Llama3 (Tuned)	GPT3.5 (Zero-shot)	GPT3.5 (CoT)	GPT3.5 (Few-shot)
S1 D ↑	**89.2**	67.4	3.7	3.7	23.2
S2 D ↑	**66.3**	56.6	3.7	3.7	11.1
Marked ↓	35.7	20.1	**1.3**	2.1	10.8

- **Findings:**
  - ChatGPT showed lower vulnerability detection rates compared to open-source LLMs (e.g., Llama3, CodeBERT).
  - Best performance was in few-shot prompting, but still significantly lower (23.2% detection).

Discussion Implications

- **LLMs for repo-level detection:**
  - Significant potential to outperform SAST tools with further refinement.
  - Generic techniques were used, meaning there's room for specialized approaches to improve performance.
- **Marked function ratios:**
  - LLMs tend to detect more vulnerabilities but with high false positives (high marked function ratios).
  - Combining multiple LLMs helped reduce false positives but requires further optimization.
- **Hybrid approaches:**
  - Combining SAST tools and LLMs leverages strengths of both approaches.
  - Can improve overall vulnerability detection and mitigate their weaknesses.

Discussion Threats to Validity

- **Internal Validity:**
  - Official implementations of tools/models were used to ensure correctness.
  - Code and data are publicly accessible to promote transparency and reproducibility.
- **Benchmarking concerns:**
  - There might be undiscovered vulnerabilities in the datasets, but the goal was to assess known vulnerabilities.
- **Data Leakage:**
  - LLMs could have been pre-exposed to some datasets during pre-training.
  - However, performance with fine-tuning vs zero-shot suggests limited memorization.

Archaeologist

Shayan Shabihi

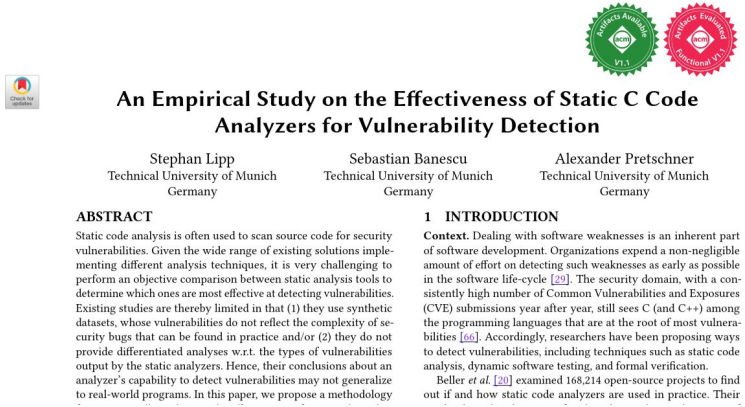
Review of Background

Particularly interesting prior work:

[51] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. 544–555.

How [51] impacted the current paper:

1. Used as the C dataset of real-world vulnerabilities
2. Referred to for selection of 5 C-friendly SAST tools
3. Used for experimental setup
   a. [51] introduces 4 detection scenarios $S_{1\text{-}1,2}$, $S_{2\text{-}1,2}$
   b. Current paper uses scenarios $S_{1\text{-}1}$, $S_{2\text{-}1}$
4. Referred to for evaluation metrics



An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection

Stephan Lipp, Technical University of Munich, Germany
Sebastian Banescu, Technical University of Munich, Germany
Alexander Pretschner, Technical University of Munich, Germany

ABSTRACT

Static code analysis is often used to scan source code for security vulnerabilities. Given the wide range of existing solutions implementing different analysis techniques, it is very challenging to perform an objective comparison between static analysis tools to determine which ones are most effective at detecting vulnerabilities. Existing studies are thereby limited in that (1) they use synthetic datasets, whose vulnerabilities do not reflect the complexity of security bugs that can be found in practice and/or (2) they do not provide differentiated analyses w.r.t. the types of vulnerabilities output by the static analyzers. Hence, their conclusions about an analyzer's capability to detect vulnerabilities may not generalize to real-world programs. In this paper, we propose a methodology

1 INTRODUCTION

Context. Dealing with software weaknesses is an inherent part of software development. Organizations expend a non-negligible amount of effort on detecting such weaknesses as early as possible in the software life-cycle [29]. The security domain, with a consistently high number of Common Vulnerabilities and Exposures (CVE) submissions year after year, still sees C (and C++) among the programming languages that are at the root of most vulnerabilities [66]. Accordingly, researchers have been proposing ways to detect vulnerabilities, including techniques such as static code analysis, dynamic software testing, and formal verification.

Beller *et al.* [20] examined 168,214 open-source projects to find out if and how static code analyzers are used in practice. Their

Review of Background

Summary of What [51] Did:

- Evaluate effectiveness of static code analyzers for C vulnerability detection (SAST tools)
- Introduces a benchmark dataset for static vulnerability analysis in C
- Assess 6 popular static analyzers on a benchmark of real-world C projects/vulnerabilities
- Proposes automatic methodology based on CWE reports to construct ground truth benchmark
- Find that tools missed 45-80% of vulnerabilities in the benchmark under best-case assumptions
- Detected vulnerabilities varied significantly depending on vulnerability subcategory
- Conclude static analyzers are limited and combining tools can help but not solve problem

How the Current Paper Differs?

- Evaluates more tools/models (18 tools, 12 LLMs vs 6 tools in [51])
- Considers additional languages (Java and Python vs just C in [51])
- Introduces repo-level formulation in addition to function-level ([51])

Review of Citations

The single paper currently having cited this paper:

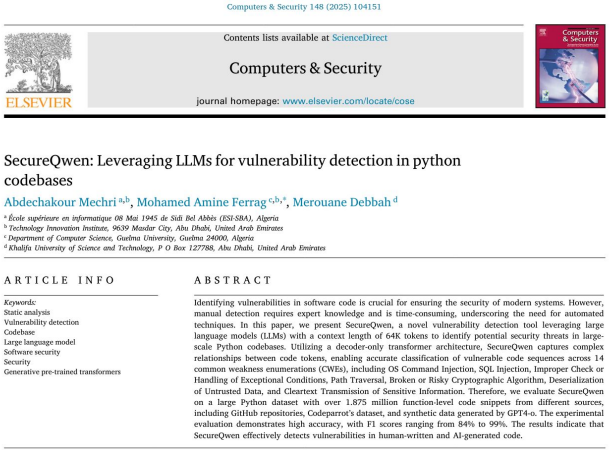
Mechri, Abdechakour, Mohamed Amine Ferrag, and Merouane Debbah. "SecureQwen: Leveraging LLMs for vulnerability detection in python codebases." Computers & Security (2024): 104151.

What is SecureQwen and what does it do?

- Is a vulnerability detection tool for Python codebases
- It fine-tunes the CodeQwen LLM (from Qwen) to classify vulnerabilities
- It introduces a new dataset (*PythonVulnDB*) of over 1.875 million Python code snippets from sources like GitHub and synthetic data
- SecureQwen evaluates the CodeQwen model on this dataset to detect 14 common vulnerability types
- It achieves high accuracy in vulnerability detection, with F1 scores ranging from 84-99%

How Does SecureQwen Use the Current Paper (Zhou et al.)?

- SecureQwen cites Zhou et al. as part of their introduction for
  a. Introduction to detecting vulnerabilities at the repository level
  b. Evaluation of different approaches like SAST and LLMs on datasets for Java, C, and Python vulnerability detection
- They used it as context for comparing their work with prior work



Computers & Security 148 (2025) 104151

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

SecureQwen: Leveraging LLMs for vulnerability detection in python codebases

Abdechakour Mechri [a,b], Mohamed Amine Ferrag [c,b,*], Merouane Debbah [d]

[a] École supérieure en informatique 08 Mai 1945 de Sidi Bel Abbès (ESI-SBA), Algeria
[b] Technology Innovation Institute, 9639 Masdar City, Abu Dhabi, United Arab Emirates
[c] Department of Computer Science, Guelma University, Guelma 24000, Algeria
[d] Khalifa University of Science and Technology, P O Box 127788, Abu Dhabi, United Arab Emirates

ARTICLE INFO

Keywords:
Static analysis
Vulnerability detection
Codebase
Large language model
Software security
Security
Generative pre-trained transformers

ABSTRACT

Identifying vulnerabilities in software code is crucial for ensuring the security of modern systems. However, manual detection requires expert knowledge and is time-consuming, underscoring the need for automated techniques. In this paper, we present SecureQwen, a novel vulnerability detection tool leveraging large language models (LLMs) with a context length of 64K tokens to identify potential security threats in large-scale Python codebases. Utilizing a decoder-only transformer architecture, SecureQwen captures complex relationships between code tokens, enabling accurate classification of vulnerable code sequences across 14 common weakness enumerations (CWEs), including OS Command Injection, SQL Injection, Improper Check or Handling of Exceptional Conditions, Path Traversal, Broken or Risky Cryptographic Algorithm, Deserialization of Untrusted Data, and Cleartext Transmission of Sensitive Information. Therefore, we evaluate SecureQwen on a large Python dataset with over 1.875 million function-level code snippets from different sources, including GitHub repositories, Codeparrot's dataset, and synthetic data generated by GPT4-o. The experimental evaluation demonstrates high accuracy, with F1 scores ranging from 84% to 99%. The results indicate that SecureQwen effectively detects vulnerabilities in human-written and AI-generated code.

Academic Researcher

Srividya Ponnada

Current Paper Analysis

SAST vs. LLMs for Repo-Level Vulnerability Detection

Significance: First comparative study on vulnerability detection using Static Application Security Testing (SAST) tools vs. Large Language Models (LLMs).

Findings:

- SAST tools: Reliable but low detection rates.
- LLMs: High detection rates, but too many false positives.
- The combination of SAST and LLMs mitigates their individual shortcomings, providing a balanced approach.

Contribution: Introduced repo-level vulnerability detection, creating a broader, more practical scope for detecting vulnerabilities in real-world repositories.

Challenges and Opportunities

High False Positives in LLMs: While LLMs detect many vulnerabilities, they flag an excessive number of non-vulnerable functions as potential threats, leading to inefficiency.

SAST Limitations: SAST tools are precise but often miss vulnerabilities.

Opportunities: The need for a system that not only detects vulnerabilities but does so with precision, minimizing false positives while maintaining high detection rates.

Maybe utilize contextual data?
Leveraging additional data such as code dependencies, historical vulnerabilities, and developer inputs could significantly enhance the precision of vulnerability detection models.

Proposed Follow-Up Project

A Context-Aware Hybrid Vulnerability Detection Framework

Objective: Develop a *Context-Aware Hybrid Framework* that combines the detection capabilities of LLMs with the precision of SAST tools, enhanced with contextual information to reduce false positives.

Components:

- **Adaptive Learning**: A feedback loop where the system learns from developers' actions (e.g., marking false positives) to improve future predictions.
- **Context Integration**: Incorporate code metadata, dependency analysis, and commit history to refine LLM predictions and reduce false alarms.
- **Real-Time Detection Pipeline**: Enable continuous learning and real-time vulnerability updates in code repositories by integrating the framework with CI/CD pipelines.

Impact: This project would result in a system for detecting software vulnerabilities, balancing detection rates with precision.

Supporting Literature, Theoretical Basis

Adaptive Learning and Feedback Loops:

Guo et al. (2021) discuss *self-improving machine learning models* through continuous feedback loops, which are essential for adapting predictions based on developer input in software security (CSATTLLM). This supports the adaptive learning component of the proposed hybrid system, which improves over time with developer interactions.

Contextual Data in Software Security:

Rahman et al. (2019) emphasizes *the importance of context (e.g., dependencies, historical vulnerabilities)* in improving prediction models for software defects. This supports the idea of integrating contextual information, including code structure, version control metadata, and software dependencies, into LLM predictions to reduce false positives.

Ensemble Methods and Hybrid Approaches:

Xu et al. (2020) proposes *ensemble learning methods* in vulnerability detection by combining static analysis tools with machine learning models for improved accuracy. This validates the idea of combining SAST tools and LLMs to balance detection precision and reduce false alarms.

Real-Time Detection in CI/CD Pipelines:

Chen et al. (2022) proposes *automating security checks* within CI/CD pipelines for continuous vulnerability detection. It emphasizes the importance of integrating real-time detection into software development workflows.

References

[1] Guo, X., Zhang, H., & Liang, C. (2021). "Adaptive Learning for Vulnerability Detection in Software Systems". *IEEE Transactions on Software Engineering*.

[2] Rahman, F., & Tantithamthavorn, C. (2019). "Defect Prediction in Large-Scale Software Systems: Using Contextual Information". *Empirical Software Engineering*.

[3] Xu, Y., et al. (2020). "Hybrid Models for Improving Software Vulnerability Detection". *ACM International Conference on Software Security*.

[4] Chen, L., & Wang, S. (2022). "Automating Security in CI/CD Pipelines: Vulnerability Detection and Mitigation". *Journal of Automated Software Engineering*.

Industry Practitioner

Paul Zaidins

The Product

- Automated repo-level vulnerability detection can save effort and catch unseen errors
  - We want to detect every vulnerability (few false negatives) and not flag genuinely innocuous code (few false positives)
  - The former catches potentially unnoticed errors and the latter saves effort
- An ensemble system for repo-level vulnerability detection
  - Boost available LLM and SAST performance with minimal effort on our part
  - Ideally a plug-and-play system (with minimal wrappers for adaptation) where we allow for any ensemble of LLM and SAST with tunable voting parameters and methods
  - When using equal voting and only-LLM or only-SAST we have systems equivalent to what is found in the paper which is proven to decrease false negatives when using only-SAST and decrease false positives when using only-LLM
  - Additional development can optionally be put in to investigating better voting systems such as allowing for weights or model confidence (in the case of LLM)

Pros and Cons

- Pros
  - Low effort boost to repo-level vulnerability detection as no new LLM or SAST are developed
  - Easy system updating by updating individual components
  - Proven performance gain using the methods outlined in the paper
  - Potentially even bigger boost by investing in testing of different ensemble voting/aggregation mechanisms
  - Theoretically can run individual components in parallel as they are independent so with optimal computational resources the inference speed is the speed of the slowest component (plus trivial voting speed)
- Cons
  - Unknown cost of creating adaptation wrappers for LLM and SAST (presumed low)
  - More models means more compute and memory are needed
  - Investigating voting methods potentially limitless cost with no guaranteed return

Takeaway

- Implementing the system as done in the paper requires minimal effort and provides proven performance gain for repo-level vulnerability detection
- Extending the system to allow for any mixture of LLMs and SASTs (and any voting system) could provided even greater performance, but this would take additional effort with likely, but not guaranteed gains

Social Impact Assessor

Tianyi Xiong

Positive Impact

- It addresses the critical issue of software vulnerabilities, posing **significant security challenges** and **potential risks to society**.
- **Repo-level vulnerability detection** task is more practical than the traditional function-level vulnerability detection. This paper has laid the groundwork for a shift in vulnerability detection efforts that can have a significant impact on real-world software security.
- Findings on the pros and cons of SAST tools, LLMs, and combining them provides valuable insights into this research field, helping researchers and practitioners to develop more effective, general and autonomous vulnerability detection strategies.

Negative Impact

- The use of API-based large language models (LLMs) for vulnerability detection may lead to data leakage. (e.g. LLM might memorize the confidential information of the code repos)
- LLM detectors might be highly sensitive to prompt injection attacks, and are less transparent compared to the SAST tools.
- This research may also inspire software attackers to design more powerful and less detectable software vulnerabilities by leveraging prior knowledge of existing detectors.

Hacker

Ethan Baker

Research Question / Goal

Goal: Test SAST tools and LLM prompting techniques on a new dataset

RQ: How effective are SAST tools and LLMs at identifying vulnerabilities in the ***ARVO*** dataset?

Related Work

1. Comparison of Static Application Security Testing Tools and Large Language Models for Repo-level Vulnerability Detection
2. ARVO: Atlas of Reproducible Vulnerabilities for Open Source Software

Tools

ARVO

- Reported vulnerabilities from OSS-Fuzz made reproducible
- >5000 pairs of secure and vulnerable docker images
- Mostly C++ projects

SEMGREP

- Open source SAST tool for analyzing C++ (and other languages)
- Simple to install
- Simple CLI

GEMINI

- Google's LLM
- Large context window so whole files can be analyzed
- Free API usage

Experimental Setup

1. Filter out complicated cases from ARVO (multiple changed files/ functions)
    a. Choose a subset of these remaining cases
    b. Get the source code for these cases using pydriller
2. Download the docker container and analyze the project with semgrep
3. Have Gemini analyze the source code for the vulnerable file
4. Compute the metrics described in the paper

Experimental Setup

1. Filter out complicated cases from ARVO (multiple changed files/ functions)
    a. Choose a subset of these remaining cases
    b. Get the source code for these cases using pydriller
2. Download the docker container and analyze the project with semgrep
3. Have Gemini analyze the source code for the vulnerable file
4. Compute the metrics described in the paper

Code and results available here: drive.google.com/drive/folders/17BPc_hOCe6vrxAvKHvHpT_sbM0KGfKyu?usp=sharing

Sample Outputs

Gemini Zero Shot

The code you provided is vulnerable to several memory-related issues that could lead to crashes or security vulnerabilities. Here's a breakdown of the vulnerable functions and why they're problematic:

1. `xsltNewAttrVT`:

- **Vulnerability:** Potential buffer overflow in the `xmlMalloc` call. The code allocates memory based on a fixed size (`MAX_AVT_SEG`) without considering the actual number of segments needed.

...

Gemini Chain of Thought

Let's break down the C++ code to identify potential vulnerabilities.

Vulnerability: Buffer Overflow in `parse_hname2` Function

The main vulnerability lies in the `parse_hname2` function. The code uses `HAVE(bytes)` macros to check for buffer overflows, but these checks are not comprehensive and can be bypassed.

Explanation:

1. **Unbounded Loop:**

...

Semgrep

gpac/src/utils/configfile.c

Avoid using 'strtok()'. This function directly modifies the first argument buffer, permanently erasing the delimiter character. Use 'strtok_r()' instead.
Details: https://sg.run/LwqG

```
572 ┆ subKeyValue = strtok((char*) keyValue,";");
    ┆ ----------------------------------------
580 ┆ subKeyValue= strtok (NULL, ";");
```

...

Results

- Gemini performed much better than expected
- Semgrep did not identify any vulnerabilities
- This experiment replicated the paper's finding of LLMs producing significantly more false positives than SAST tools

Case	semgrep	Gemini ZS	Gemini COT
52901	0/0	1/5	0/5
49654	0/0	1/5	1/5
57234	0/0	1/6	1/6
44766	0/0	0/0	0/5
39802	0/0	1/1	1/1
S1	0/5	4/5	3/5
Marked Functions	0	17	22

Next Steps

- It is likely that there is some test set contamination when using old vulnerability datasets. Transforming the data to address this may provide more insight
- Only giving the model the vulnerabile file is not applicable to the real world. Redoing this experiment “repo wide” makes more sense.
- Semgrep may not be a good fit for these projects or vulnerabilities. Using more varied SAST tools may be valuable.

Private Investigator

Mohammed Afaan Ansari

Xin ZHOU - First Author

- Affiliation: Singapore Management University, Singapore
- Educational Background: PhD candidate in computer science
- Motivation: Xin Zhou is motivated by the need for automated vulnerability detection tools. With her background in AI and security, this project extends her work on using LLMs to improve detection accuracy and efficiency.
- Previous Project: Involved in research related to software security and machine learning applications in code analysis.
  i.e. Large language model for vulnerability detection: Emerging results and future directions

Thanh Le-Cong

Education

University of Melbourne
Doctor of Philosophy - PhD, Computer Software Engineering
Feb 2023 - Aug 2026

Hanoi University of Science and Technology
Bachelor's degree (Talented Engineering Program), Information Technology
2016 - 2021

Experience

Graduate Researcher
University of Melbourne · Full-time
Feb 2023 - Present · 1 yr 9 mos
Melbourne, Victoria, Australia

Applied Scientist II
Amazon Web Services (AWS) · Internship
Jun 2024 - Sep 2024 · 4 mos
Arlington, Virginia, United States · On-site

Research Engineer
Singapore Management University · Full-time
Jan 2022 - Dec 2022 · 1 yr
Singapore

Undergraduate Research Assistant
Hanoi University of Science and Technology · Part-time
Jun 2017 - Dec 2021 · 4 yrs 7 mos
Hanoi, Vietnam

Undergraduate Research Assistant
Griffith University · Full-time
Jun 2019 - Jun 2019 · 1 mo
Queensland, Australia

Thanh Le-Cong

- Motivation: Thanh Le-Cong is motivated by the potential of using AI tools like LLMs to improve software vulnerability detection and bridge gaps between traditional security tools / testing methods and AI-driven solutions
- Previous Project: Involved in prior research exploring machine learning application in software engineering.
  i.e. Refining chatgpt-generated code: Characterizing and mitigating code quality issues