Journal of Information Security, 2025, 16(2), 341-357
https://www.scirp.org/journal/jis
ISSN Online: 2153-1242
ISSN Print: 2153-1234

Large Language Models (LLMs) for Software Security Analysis

Brian Mata, Vijay Madisetti

Georgia Institute of Technology, Atlanta, USA
Email: bmata3@gatech.edu, vkm@gatech.edu

How to cite this paper: Mata, B. and Madisetti, V. (2025) Large Language Models (LLMs) for Software Security Analysis. *Journal of Information Security*, **16**, 341-357.
https://doi.org/10.4236/jis.2025.162018

Received: March 24, 2025
Accepted: April 26, 2025
Published: April 29, 2025

Copyright © 2025 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/
Open Access

Abstract

Security vulnerabilities are a widespread and costly aspect of software engineering. Although tools exist to detect these vulnerabilities, non-machine learning techniques are often rigid and unable to detect many types of vulnerabilities, while machine learning techniques often struggle with large codebases. Recent work has aimed to combine traditional static analysis with machine learning. Our work enhances this by equipping LLM-based agents with classic static analysis tools, leveraging the strengths of both methods while addressing their inherent weaknesses. We achieved a false detection rate of 0.5696, significantly improving over the previous state-of-the-art LLM-enabled technique, IRIS, which has a false detection rate of 0.8482. Furthermore, using Claude Sonnet 3.5, our technique produces an F1 score of 0.1281, which is an improvement over the standard CodeQL suite and approaches IRIS's score of 0.1770.

Keywords

Large Language Models, Software Security

1. Introduction

Security vulnerabilities are pervasive in software. It is estimated that up to 98% of software contains a security vulnerability that could harm users [1]. Furthermore, up to 91% of web software is vulnerable to data breaches [1], which could cost companies an average of 4.88 million USD as of 2024 [2]. Despite the prevalence and cost of security vulnerabilities, detecting software defects that could lead to these vulnerabilities remains an open problem. Current methods have drawbacks, such as a high rate of false positives and an inability to detect a wide range of CWEs [3].

However, modern large language models have made significant advances in code understanding through tasks such as program repair [4] and bug detection [5]. Despite these advances, studies on the efficacy of LLMs for security analysis still find that LLMs struggle to reason over entire codebases in ways typical of security defects [6] [7].

2. Existing Work

The existing body of work on static application security testing is extensive. Research ranges from methods not using any machine learning to those dominated by large language models. One classic technique enumerates paths from source to sink, where sources and sinks are determined by hand-designed rules operating on bytecode [8]. There are deep learning methods that do not rely on large language models; they simply use CNNs and RNNs [9]. However, these methods are limited to a small subset of security vulnerabilities, and the authors express that more work is needed.

Recently, techniques leveraging pre-trained LLMs have emerged. An early technique involved directly querying the LLM with a code snippet to determine if it contained a vulnerability [10]. The authors noted that ChatGPT showed impressive results, but its ability to detect vulnerabilities directly was still inferior to prior SAST methods. The authors of DiverseVul [11] made a similar observation about the LLMs they trained for this task: good performance, but not good enough. The consensus is that LLMs alone are not yet ready for this use case [7], as shown in **Figure 1**.

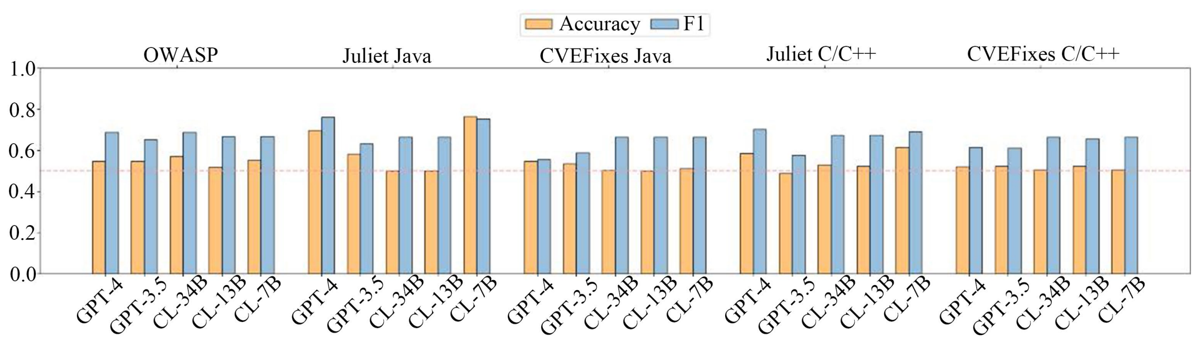

Figure 1. A comparison between LLMs at detecting vulnerabilities across datasets shows authors noting low F1 scores, even with frontier models.

Recently, efforts have combined traditional methods that analyze code structure with LLM methods to better understand that code. One such effort is the IRIS framework [12], which combines LLM techniques with CodeQL [13]. This approach operates similarly to classical techniques, identifying sources and sinks and detecting unsanitized paths between them. However, it uses LLMs to label potential sources and sinks and to prune false positives when identifying unsanitized paths from a source to a sink. This approach leverages the strengths of classical techniques in scanning an entire code base while utilizing LLMs' ability to understand isolated snippets.

3. Proposed Approach

This approach builds on the IRIS framework [12] by incorporating a recent technique from general static analysis, the Intelligent Code Analysis Agent (ICAA) [14]. We aim to deconstruct the IRIS framework into its core components, using CodeQL to scan the codebase and LLMs to determine which parts to scan. We combine this with the ReAct agent from the ICAA paper. To our knowledge, this is the first agent-based model used solely for static security analysis. A diagram of this architecture can be seen in **Figure 2**. We expand on the IRIS paper in two ways. First, our agent's behavior encompasses all of IRIS's capabilities, as it has access to all its constituent parts. Additionally, we leverage LLMs to guide traditional static analysis tools for a better understanding of extensive codebases, addressing a traditional weakness of machine learning techniques. LLMs can write CodeQL queries directly to refine their search of the entire codebase.

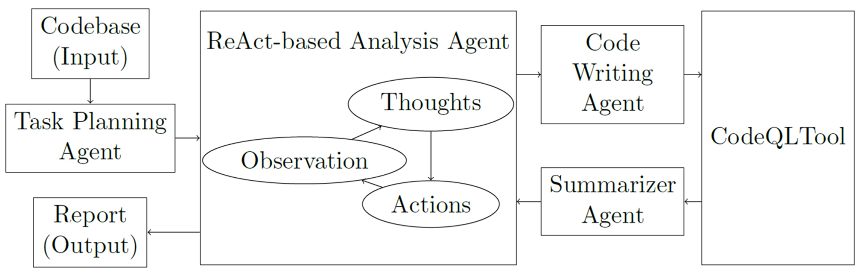

Figure 2. Adapted ICAA framework.

To measure the progress of our work, we use the CWE-Bench from the Li *et al.* paper [12]. It consists of a set of Java repositories and CVE-fixing commits. Additionally, it labels a set of vulnerabilities.

4. Detailed Design

At a high level, this framework takes a description of the codebase, uses it to generate a plan to employ CodeQL for understanding the codebase, and detects vulnerabilities. This plan is sent to a ReAct-style agent that can generate a thought, choose to generate a CodeQL expression, or report a vulnerability. Due to the complexity of writing CodeQL expressions, we delegate this task to an agent separate from the ReAct loop. This agent tries to generate an expression that fulfills the ReAct agent's request. Once it has a query that can run and produce results, those results are passed to a summarizer agent. A summary is returned to the ReAct agent, which continues operating until it reaches its turn limit or finishes finding vulnerabilities. Once completed, the ReAct agent returns all reported vulnerabilities as a JSON report.

4.1. Task Planning

Since it's impossible to pass the entire codebase into an LLM to generate a task plan, the task planning agent divides this into two steps: generating a code briefing and then using the code briefing to generate a task plan.

4.1.1. Code Briefing

To generate the briefing, the agent starts by scraping the codebase, examining the file structure and the README file. We then pass this information to an LLM, which creates a brief summary of the codebase with instructions to focus on potential security concerns. Snippets from a sample codebase summary are included in Figure 3.

```
This appears to be a Jenkins plugin repository for Docker Commons API.
Here's a summary of key insights:

...

Security Considerations:
1. The plugin handles credentials for both Docker Registry and Docker Daemon access,
making it a sensitive component from a security perspective
2. There's a CodeQL setup present (custom_codeql_build.sh and code_ql_database
directory) suggesting security scanning is in place
3. The example pipeline code shows credential handling using the DOCKER_CERT_PATH
environment variable
4. The plugin uses credential binding which, if implemented incorrectly, could
potentially expose sensitive information
```

Figure 3. A snippet from a sample codebase summary from the codebase summarizing step of the task planning agent.

4.1.2. Task Plan

Given this summary, we ask the task planning agent to generate a plan. The strategy used is to assign each agent a software engineer of a given level using the system prompt. For the task plan, we inform the LLM that it is a principal engineer hosting office hours. We then prompt it with the code briefing, asking how it would use CodeQL to identify security flaws.

```
* Map Data Flow and Trust Boundaries
  * Query for all external API calls using 'External::getACallSite()'
  * Track data flow from user inputs to sensitive operations using
    'DataFlow::Configuration'
  * Identify all network socket operations and HTTP client usage
  * Map file system interactions to understand file access patterns
  * Find all database queries and data persistence operations

* Analyze Cross-Language Interactions
  * Trace data flow across language boundaries (Java to .NET)
  * Identify shared resource access between different language components
  * Map inter-process communication patterns
  * Find serialization/deserialization points between language components
  * Locate shared configuration file usage across language boundaries
...
```

Figure 4. A snippet of the task plan generated by the task-planning agent.

Since this output is for our ReAct agent, which can be long-running and quickly reach its context limit, we are mindful of the number of tokens consumed by this plan. To create a concise summary, we generate a bulleted list with sections. This is achieved by specifying a tool that requires the LLM to return valid JSON with a schema, which is then formatted into a bulleted list. A snippet of a sample plan can be seen in **Figure 4**.

4.2. ReAct Agent

The ReAct agent [15] is central to this technique. The intention of this agent is to create an agent that can write CodeQL expressions, analyze the results, and make follow-up queries based on the results. To facilitate this, when generating a response, it can choose from a list of four possible tools. The tools, their required fields, and a brief description of how the LLM should fill those fields are provided below:

- Thought
  - Description: Explain your thoughts and the next steps you plan to take in as much detail as possible, using clear and straightforward language.
- CodeQL
  - Description: Provide a comprehensive overview of a CodeQL query you wish to execute on the codebase. Avoid including any code. Instead, offer a detailed explanation to assist a junior developer, who may not be familiar with CodeQL, in crafting the query.
- Report Vulnerability
  - Filepath: The primary filepath where the vulnerability exists
  - Explanation: A brief overview of how the vulnerability works
  - CWE: The CWE that most closely matches the detected vulnerability
- No More Vulnerabilities
  - Recap: Provide a recap of the work completed by the agent so far to determine if it can be terminated early when no longer useful.

```
assistant:
[12/50] Find methods in the SCP implementation that handle file paths by:
1. Looking for methods in classes under the sshd-scp directory
2. Methods that take String parameters representing file paths
3. Methods with names containing 'path', 'file', 'directory', or 'transfer'
4. Focus on methods that process or validate these paths before file operations

For each match, show:
- The method name and class
- The parameter types (especially String parameters that might be paths)
- Any path manipulation or validation logic
- The location in the codebase

This will help identify potential path traversal vulnerabilities in the SCP
implementation.
```

Figure 5. A sample output of the agent using the CodeQL tool to describe a CodeQL query it would like to run.

When the agent generates a thought or reports a vulnerability, we add it to the history and pass it back to the agent for another response. When the agent chooses the CodeQL tool, we send the description to a separate code-writing agent who returns a valid CodeQL expression. We summarize these results and pass the summary back to the ReAct agent. Each request for the agent to generate more text from the LLM constitutes a turn. We limit the number of turns to prevent infinite loops. This parameter can be adjusted to balance precision and recall. To inform the LLM agent about its turn limit and encourage progress, we prepend a turn counter to each input (e.g., 12/50). A sample turn of the agent is shown in Figure 5.

We instruct the LLM, acting as a senior engineer, to assist junior engineers in using CodeQL to identify security flaws. This approach encourages the LLM to be more descriptive, aiding in progress. We provide the task plan from the task planning phase.

4.3. Code Writing Agent

We noticed that even advanced models struggle to produce valid CodeQL on their first attempt. This is likely because CodeQL is not a popular language in terms of online resources during the training time of the LLM. However, passing the errors back to the LLM often enables it to generate a valid CodeQL expression. This process is expensive in terms of the number of tokens spent going back and forth with the LLM, which pollutes the ReAct agent's history. To remedy this, we offload the code-writing process to a separate agent that only needs a plain English description of a CodeQL query.

```
class RequestHandler extends Method {
  RequestHandler() {
    exists(Annotation a | a = this.getAnAnnotation() |
      a.getType().getName() in [
        "RequestMapping", "GetMapping", "PostMapping",
        "PutMapping", "DeleteMapping", "PatchMapping" ]
    ) or
    exists(Parameter p | p = this.getAParameter() |
      p.getType() instanceof ServletRequest or
      p.getType().getName() in ["HttpServletRequest", "WebRequest",
      "NativeWebRequest"]
    ) or
    exists(Call ma |
      ma.getCallee().getName() in ["getParameter", "getParameterValues",
      "getParameterMap"] and
      ma.getEnclosingCallable() = this
    ) or
    exists(Parameter p | p = this.getAParameter() |
      p.getName().toLowerCase() in ["path", "filepath", "url", "uri"] or
      p.getType().getName() in ["Path", "File", "URI", "URL"])
  }
}
from RequestHandler handler
select handler,
  "HTTP request handler method in " + handler.getDeclaringType().getName() +
  " with signature: " + handler.getSignature()
```

Figure 6. A sample valid CodeQL expression generated by the code-writing LLM agent.

In this agent, we pass errors back to the LLM until it generates a valid CodeQL expression. Once we have a valid expression, the task is not complete. An issue can arise when a CodeQL produces results that exceed the LLM's context window limit. In such cases, the results might be too long to return to the ReAct agent. To address this, we allow the code-writing agent to run the CodeQL query and count the number of tokens. If it exceeds a certain threshold, we return the task plan to the code-writing agent and ask it to write a query that reduces the number of results. A sample valid CodeQL generated by the agent is provided in Figure 6.

4.4. Summarizer Agent

For many of the reasons provided earlier, simply piping the CodeQL results back into the ReAct agent often encounters problems with the context-window limit. Therefore, a simple LLM query is run to summarize the results before returning them to the ReAct agent. A sample output can be seen in Figure 7.

```
...

Security-relevant findings:
1. Multiple key decoders and parsers that need careful review for proper
implementation
2. Several authentication-related key handlers that are security-critical
3. Session management code that handles sensitive data
4. Key generation and loading functionality that requires secure implementation

Here are some of the most interesting findings from a security perspective:

‘‘‘java
// Key authentication handlers
/sshd-core/src/main/java/org/apache/sshd/server/auth/pubkey/UserAuthPublicKey.java:73
/sshd-core/src/main/java/org/apache/sshd/server/auth/pubkey/
CachingPublicKeyAuthenticator.java:53

// Key generation and loading
/sshd-common/src/main/java/org/apache/sshd/common/keyprovider/
AbstractGeneratorHostKeyProvider.java:342
/sshd-common/src/main/java/org/apache/sshd/common/config/keys/loader/
KeyPairResourceLoader.java:152

‘‘‘
...
These findings suggest areas that would benefit from focused security review,
particularly around key handling, authentication flows, and session management.
```

Figure 7. A sample output of the summarizer agent using parsed CodeQL results.

5. Results

5.1. Experimental Setup

For this design, we chose Anthropic's Claude Sonnet 3.5 and 3.7 for the model underpinning the agent framework. We experimented with smaller models; however, these did not produce acceptable results for agent framework use. We set max_tokens to 4096 and temperature to 0.3. These values were chosen based on [16], which recommends a low temperature (less than 1.0) for structured outputs, such as code. As for the max tokens, CodeQL expressions can be quite verbose, so we allowed the

LLM to produce long outputs. We observed early stopping frequently with Claude, so we were confident it would only use the extra tokens when useful.

We set 30 turns for the ReAct agent, allowing it to use a tool before being cut off. We observed a weak negative correlation between turn count and detection rate during tests. Generally, by 30 turns, the agent completed its productive work and tended to meander afterward.

Finally, we set a 50k character limit on the result size from a CodeQL query before asking for a query that produces fewer results. Although we could specify larger limits according to our model's capabilities, we wanted to avoid a "needle in the haystack" problem, where the LLM would struggle to find useful results among many.

5.2. Dataset

To measure the efficacy of our technique, we chose the CWE-Bench-Java dataset from Li *et al.* [12]. This dataset consists of real-world Java repositories with known and patched vulnerabilities. The patches are used to annotate file names and function names of vulnerable code. This is a hand-curated and validated dataset. By relying on real-world datasets, it tests the ability of our agent framework to work with complicated codebases.

We were unable to generate CodeQL databases for all repositories. We were limited to 95 out of 120 repositories. Therefore, our results might not directly compare to those from the IRIS paper. However, we have included them here for reference.

One challenge is that using CodeQL requires an instrumented build of the Java application. While this might not be difficult for teams familiar with their builds, it is challenging for security researchers working on unfamiliar projects. Generating an instrumented build with CodeQL is also more temperamental than building the project as is (see **Appendix**).

5.3. Evaluation Metrics

We analyzed three main metrics to compare our framework's performance with previous work. For our agent, we consider a vulnerability detected if the filename, description, and reported CWE are all correct.

- **Detection Rate**

$$=\frac{\text{Number of Detected Vulnerabilities}}{\text{Number of Known Vulnerabilities}}$$

- **Average False Detection Rate (FDR)**

$$=\frac{\text{Falsely Flagged Vulnerabilities}}{\text{All Flagged Vulnerabilitie}}$$

- **Average $F1$ score**

$$=\frac{2(\text{Detected Vuln})}{2(\text{Detected Vuln})+\text{FalselyFlagged Vuln}+\text{Undetected Vuln}}$$

We measure the average false detection rate and the average $F1$ score for each repository in the dataset and then calculate the average across all repositories.

5.4. Evaluation

Given our experimental setup, we observed performance comparable to IRIS in terms of the $F1$ score. For this dataset, we exceeded both CodeQL and IRIS in the false detection rate. Combined with the small number of reported vulnerabilities per repository, this should reduce the manual work needed to vet vulnerabilities. However, IRIS detected more vulnerabilities overall than our technique. Our results are shown in Table 1.

Table 1. Comparison of static security analysis techniques. As mentioned, we cannot directly compare our results to those from the IRIS paper because we were unable to generate CodeQL databases for 25 out of 120 repositories in the dataset. Our work is labeled as ReAct agent for Static Analysis.

Method	$F1$ Score	FDR	Detection Rate
ReAct agent for Static Analysis (+ *Claude Sonnet* 3.5)	0.1281	0.8491	0.1591
ReAct agent for Static Analysis (+ *Claude Sonnet* 3.7)	0.0754	**0.5696**	0.0857
CodeQL	0.0760	0.9003	0.2250
IRIS (+ ChatGPT 4)	**0.1770**	0.8482	**0.4583**

5.4.1. Breakdown by CWE

We can break down the detected vulnerabilities by their CWE ID. Looking at Figure 8, we can see that the detection rates are relatively uniform across the different CWE classes.

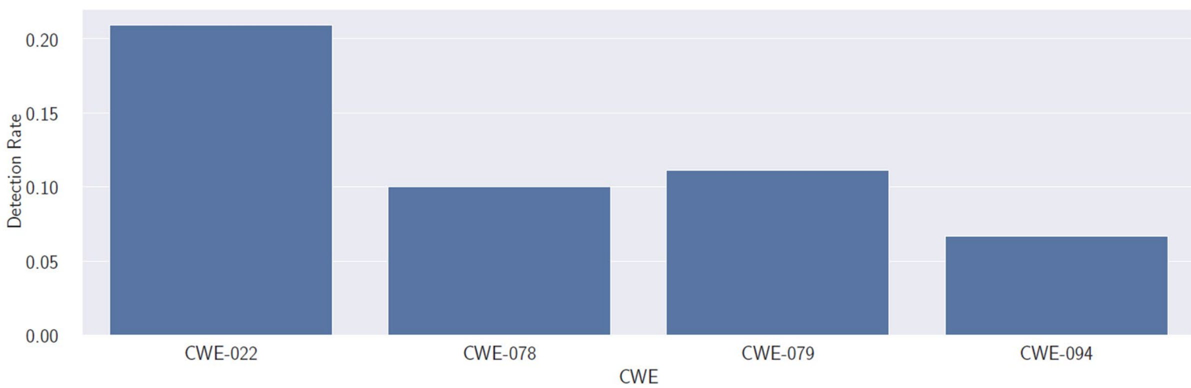

Figure 8. Detection Rate by CWE ID with the Claude Sonnet 3.5 Run.

5.4.2. Effect of Number of Turns on Performance

As mentioned above, we measure the performance of the agent framework against three criteria: $F1$ score, detection rate, and false detection rate. In Figure 9, we examine how the number of turns used by the ReAct agent affects performance as measured by these criteria.

One trend observed in this data is that the number of turns negatively correlates with performance. In all examples where our framework correctly identified vulnerabilities, the ReAct agent was terminated early via the NoMore tool.

Figure 9. Performance versus the number of turns used by the ReAct agent: We observe a weak negative relationship between the number of turns used and performance in vulnerability detection.

In **Figure 10**, we see that early termination is the most common outcome; however, the ReAct agent does not behave this way for all repositories in the dataset.

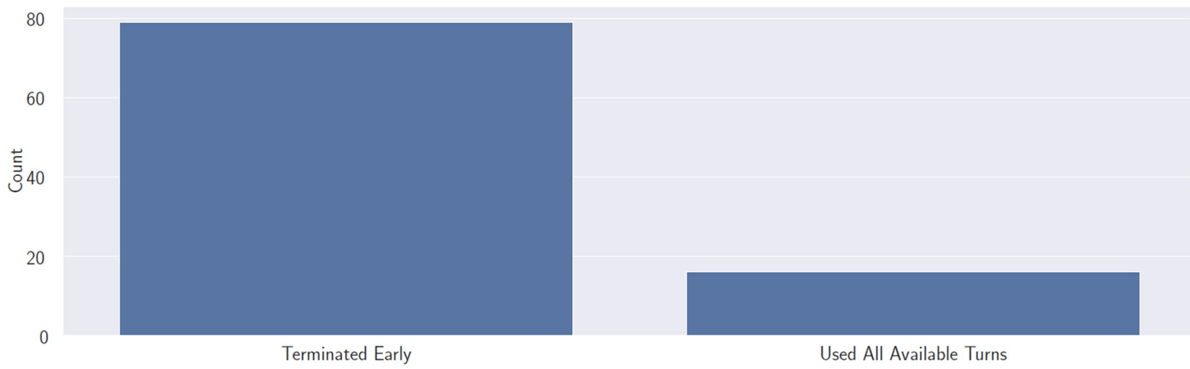

Figure 10. Comparison of the number of repositories where the ReAct agent terminated early versus those where it used all available turns.

5.4.3. Impact of The Code Writing Agent on Performance

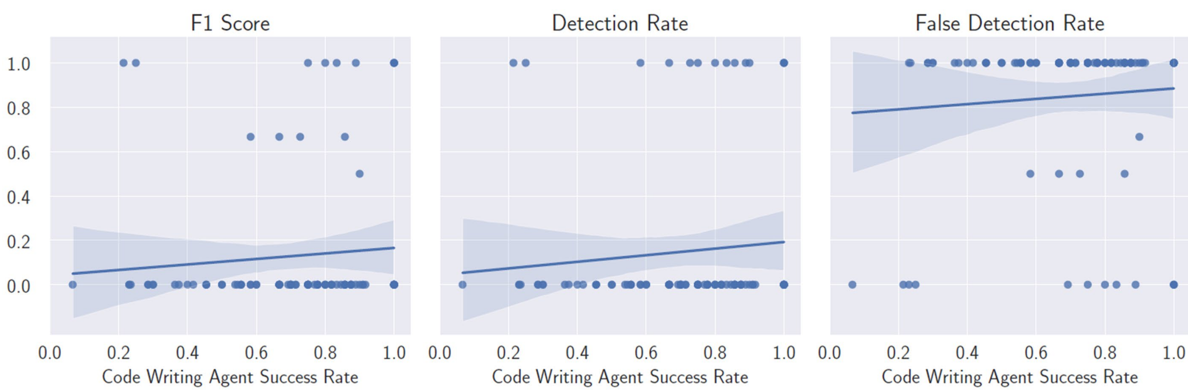

Figure 11. The code writing agent's success rate positively affects the framework's ability to detect vulnerabilities. We observe a positive relationship between the agent's success rate and the framework's performance.

One aspect that heavily influences the agent's ability to understand the code and find vulnerabilities is whether it can write the necessary CodeQL queries. In Figure 11, we see that the majority of repositories where the framework performed well are those where the code-writing agent can produce reliable code more than half of the time.

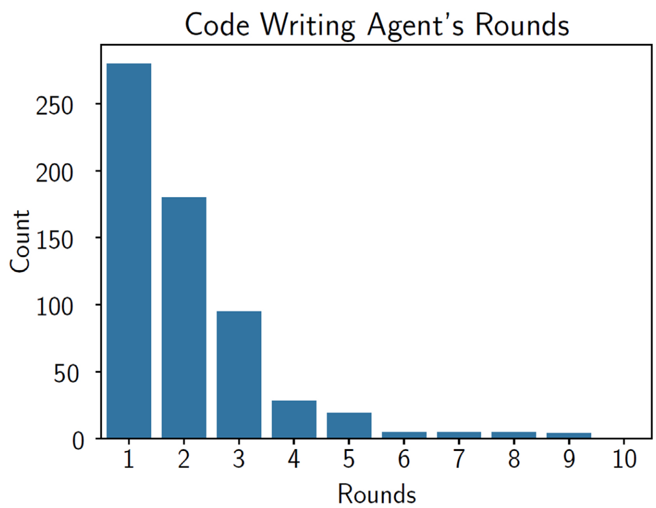

Figure 12. The distribution of the number of rounds used by the code-writing agent to produce a working query includes fixing all compilation errors in the CodeQL query, optimizing the query to run within the allotted time, and rewriting the query to prevent returning too many results if necessary. Only successful query generations are included in this chart.

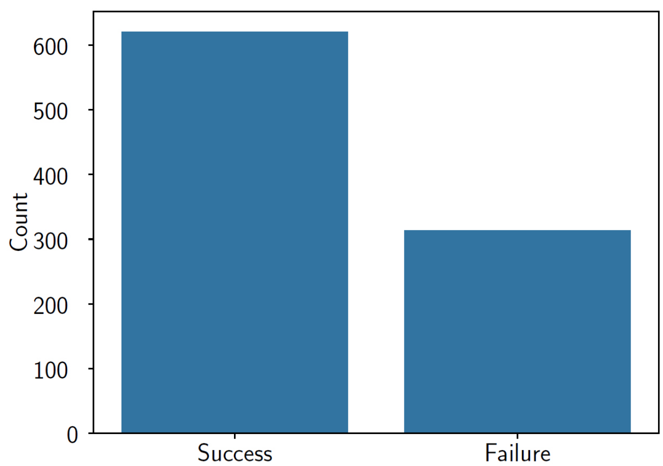

Figure 13. The number of times the code-writing agent produced a valid CodeQL query versus the times it failed to do so.

One interesting aspect of the code writing agent's performance is its binary nature: it either works or it doesn't. In Figure 12, we see the distribution of the number of rounds used by the agent to produce a valid query. In our dataset, using more than three rounds was rare in scenarios where it successfully produced a query. However, Figure 13 shows the rate at which successful queries were produced. Roughly a third of the time, the code writing agent was unable to produce a valid CodeQL query despite being given 10 rounds.

There are multiple reasons the code-writing agent could mark a round as failed. In Figure 13, we see that most errors were caused by generating code that failed to compile. Interestingly, the agent is almost never able to recover from runtime errors or cases when the output is too long. Enabling the agent to recover from these scenarios could be a topic of future research. One idea would be to use an agent with access to the CodeQL documentation to help the code-writing agent troubleshoot runtime errors (Figure 14).

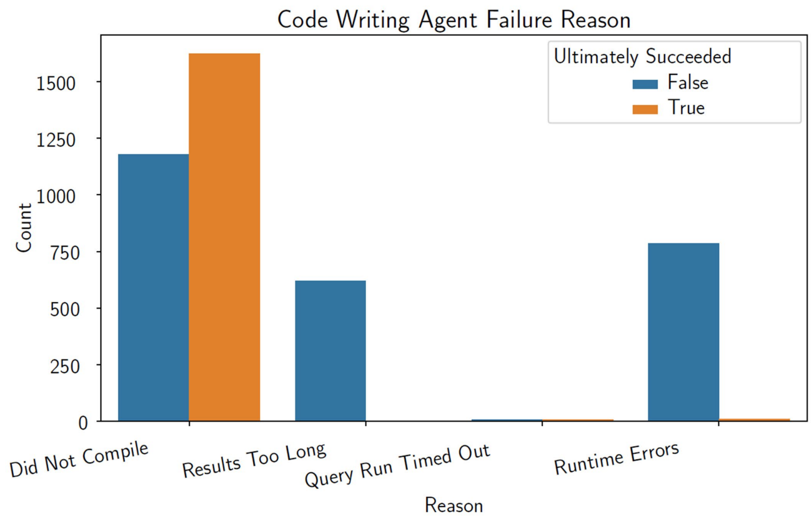

Figure 14. The number of times we observed a particular reason for the CodeQL agent failing to produce a valid expression. This counts the attempts made within the code-writing agent's retry loop. As the code-writing agent gets up to 10 rounds to produce a valid CodeQL expression, each request from the ReAct agent to the code-writing agent can result in up to 10 failures on this chart.

6. Conclusions and Future Work

This project has successfully positioned the LLM at the forefront of automated static security analysis. We demonstrated that it can write useful CodeQL queries and advance the understanding of the codebase. In bringing this idea to life, we've created a framework with a significantly improved $F1$ score compared to base CodeQL techniques, approaching the previous state-of-the-art LLM-enabled static security analysis, IRIS. Furthermore, in false detection rates, a key issue for many developers, the agent framework significantly exceeds previous research.

6.1. Future Work

6.1.1. False Pruner Agent

One key strength of this work was the relatively low rate of false positives. We believe this could be further improved by using a false pruner agent, which could further reduce the rate of false positives. One approach could involve sending the reported vulnerabilities back through the ReAct agent, allowing it to write CodeQL to scrutinize each vulnerability further.

6.1.2. Comparison of Frontier Models

After some experimentation, we decided against models like LLAMA 7b and DeepSeeker coder. We found these models were not suitable for working as agents or writing CodeQL. Ultimately, we based much of our work on Anthropic's Claude family of models. However, future work could explore comparable models from OpenAI and others.

6.1.3. Additional Tools

The design of this agent framework only allows the model to interact with the codebase via CodeQL. While this is a powerful tool, having access to the actual code might be beneficial. For example, using the cat CLI utility could be helpful, especially for finalizing reported vulnerabilities.

6.2. Analysis of LLM Data Contamination

One concern is that when working with publicly known vulnerabilities, such as those in this dataset, the LLM may have encountered information about them before. We'd like to use this agent framework, as well as the IRIS framework, to analyze private datasets—specifically, on code the LLM could not have seen before—and determine if these methods are as effective in that context.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Threats and Vulnerabilities in Web Applications 2020-2021. https://global.ptsecurity.com/analytics/web-vulnerabilities-2020-2021

[2] Cost of a Data Breach Report 2024. https://www.ibm.com/reports/data-breach

[3] Source Code Analysis Tools. https://owasp.org/www-community/Source_Code_Analysis_Tools

[4] Wei, Y., Xia, C.S. and Zhang, L. (2023) Automated Program Repair in the Era of Large Pre-Trained Language Models. *Proceedings of the* 45*th International Conference on Software Engineering* (*ICSE* 2023), Melbourne, 14-20 May 2023.

[5] Li, H., Hao, Y., Zhai, Y. and Qian, Z. (2024) Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proceedings of the ACM on Programming Languages*, **8**, 474-499. https://doi.org/10.1145/3649828

[6] Steenhoek, B., Rahman, M., Roy, M.K., Alam, M.S., *et al.* (2024) A Comprehensive Study of the Capabilities of Large Language Models for Vulnerability Detection.

[7] Khare, A., Dutta, S., Li, Z.Y., *et al.* (2024) Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities.

[8] Benjamin Livshits, V. and Lam, M.S. (2005) Finding Security Vulnerabilities in Java Applications with Static Analysis. *USENIX Security Symposium*, **14**, 18.

[9] Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., *et al.* (2018) Automated Vulnerability Detection in Source Code Using Deep Representation Learning. 2018 17*th IEEE International Conference on Machine Learning and Applications*

(*ICMLA*), Orlando, 17-20 December 2018, 757-762.
https://doi.org/10.1109/icmla.2018.00120

[10] Bakhshandeh, A., Keramatfar, A., Norouzi, A. and Chekidehkhoun, M.M. (2023) Using ChatGPT as a Static Application Security Testing Tool.

[11] Chen, Y.Z., Ding, Z.J., Alowain, L., Chen, X.Y. and Wagner, D. (2023) DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning-Based Vulnerability Detection.

[12] Li, Z.Y., Dutta, S. and Naik, M. (2024) LLM-Assisted Static Analysis for Detecting Security Vulnerabilities.

[13] Codeql (2021) GitHub Inc. https://codeql.github.com/

[14] Fan, G., Xie, X.H., Zheng, X.J., Liang, Y.N. and Peng, D. (2023) Static Code Analysis in the AI Era: An In-Depth Exploration of the Concept, Function, and Potential of Intelligent Code Analysis Agents.

[15] Yao, S.Y., Zhao, J., Yu, D., *et al.* (2023) React: Synergizing Reasoning and Acting in Language Models.

[16] (2024) LLM Temperature—The Secret Sauce to Tuning AI Responses.

Appendix

A1. Sample Correctly Detected Vulnerabilities

A1.1. Jolokia: CVE-2018-1000129

Here is a sample output where our tool was able to successfully detect a real vulnerability. Compared to the diff here, the description is correct and quite instructive.
https://github.com/jolokia/jolokia/commit/5895d5c137c335e6b473e9dcb9baf748851bbc5f#diff-f19898247eddb55de6400489bff748adGithub diff for security patch.

A1.2. Vertx-Web: CVE-2019-17640

```
"filepath": "/agent/core/src/main/java/org/jolokia/http/AgentServlet.java",
"explanation": "The application's AgentServlet handles user input from HTTP requests
and reflects it back in HTML responses without proper sanitization. This creates a
Cross-Site Scripting vulnerability where an attacker could inject malicious
JavaScript that would execute in users' browsers when they view the response. The
vulnerability appears around line 472 where user input flows directly into HTML
output generation. Despite the presence of an EscapeUtil class in the codebase, this
particular code path appears to bypass proper HTML encoding.",
"cwe": "CWE-79"
```

Figure A1. A sample vulnerability was detected by our tool in the Jolokia repository. All output is AI-generated by our LLM agent.

Compared to the description from https://bugs.eclipse.org/bugs/show_bug.cgi?id=567416 "Eclipse Vert.x StaticHandler doesn't correctly process backslashes," all fields are correct. The description, the marked CWE, and the explanation are all accurate.

```
"filepath":
    "/vertx-web/src/main/java/io/vertx/ext/web/handler/impl/StaticHandlerImpl.java",
"explanation":
    "The StaticHandlerImpl class handles serving static files and contains path
    manipulation operations that could be vulnerable to path traversal attacks. The
    class processes user-supplied paths to serve static files, and if path validation
    or normalization is not properly implemented, it could allow attackers to access
    files outside the intended directory using sequences like \"../\" to traverse up
    the directory tree. This is particularly concerning in a web server context where
    proper path validation is crucial for security.",
"cwe":
    "CWE-22"
```

Figure A2. Another sample vulnerability in the vertex web repo.

A2. Incorrectly Marked CWEs

The agent marked the wrong CWE for a few vulnerabilities but found the right file. After manual examination, we decided not to mark these as correctly identified vulnerabilities.

Jenkinsci Perfecto-Plugin: CVE-2020-2261

https://github.com/jenkinsci/perfecto-plugin

A3. Some Examples of the Code Writing Agent

A3.1. Jolokia: CVE-2018-1000129

```
"filepath":
    "PerfectoCredentials.java",
"cwe":
    "CWE-79",
"explanation":
    "The application displays credential-related information in the Jenkins UI
    without proper HTML escaping. Specifically, user-provided credential data is
    processed and rendered directly in HTML output around line 195-200. This could
    allow an attacker to inject malicious JavaScript code through credential fields
    that would then be executed in the context of other users' browsers when viewing
    the credentials page. This is particularly concerning as it appears in credential
    handling code, which is a sensitive security area where compromise could lead to
    further system access."
```

Figure A3. This description, while pointing in the right direction, seemed too loosely related to actual vulnerabilities to effectively guide developers.

A3.1.1. Prompt

```
Find potential code injection vulnerabilities by looking for:

1. Dynamic class loading operations:
   - ClassLoader usage
   - Class.forName() calls
   - URLClassLoader instantiation
   - Custom classloader implementations

2. Reflection usage:
   - Method.invoke()
   - Constructor.newInstance()
   - Field.set()
   - Any reflection API usage with dynamic values

3. Script evaluation:
   - ScriptEngine usage
   - eval() like functions
   - Dynamic code compilation

4. Focus on cases where:
   - The class name or method name comes from variables
   - The input to these operations can be influenced by external data
   - There's string concatenation or manipulation before these operations

Pay special attention to cases where user input or configuration values flow into
these dangerous operations.
```

Figure A4. The plain English description of a CodeQL query.

A3.1.2. CodeQL Output

```
import java
import semmle.code.java.dataflow.TaintTracking
import semmle.code.java.dataflow.FlowSources
import DataFlow::PathGraph

class DangerousCodeExecution extends Method {
  DangerousCodeExecution() {
    // Class loading
    this.getDeclaringType().hasName("Class") and this.hasName("forName") or
    this.getDeclaringType().hasName("ClassLoader") and this.hasName("loadClass") or
    // Reflection
    this.getDeclaringType().hasName("Method") and this.hasName("invoke") or
    this.getDeclaringType().hasName("Constructor") and this.hasName("newInstance") or
    this.getDeclaringType().hasName("Field") and this.hasName("set") or
    // Script evaluation
    this.getDeclaringType().getASupertype*().hasName("ScriptEngine") or
    this.hasName("eval") or
    this.getDeclaringType().hasName("JavaCompiler") and this.hasName("compile")
  }
}
class DynamicCodeConfig extends TaintTracking::Configuration {
  DynamicCodeConfig() { this = "DynamicCodeConfig" }
  override predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }
  override predicate isSink(DataFlow::Node sink) {
    exists(Call call |
      call.getCallee() instanceof DangerousCodeExecution and
      sink.asExpr() = call.getAnArgument()
    )
  }
}
from DataFlow::PathNode source, DataFlow::PathNode sink, DynamicCodeConfig conf
where conf.hasFlowPath(source, sink)
select sink.getNode(), source, sink, "Potentially unsafe dynamic code execution with
    input from $@.",
  source.getNode(), "user input"
```

Figure A5. CodeQL output.

A3.1.3. Analysis of CodeQL Results

```
user:
[13/30] Let me analyze this CodeQL output which is searching for potential Cross-Site
Scripting (XSS) vulnerabilities.

Summary:
- The query identifies paths where untrusted user input could flow into HTML output
without proper sanitization
- Most findings are centered in the AgentServlet.java file, which appears to be a
core component handling HTTP requests
- There are also potential issues in supporting utility classes like EscapeUtil.java,
IoUtil.java, and LogHandler.java
- Multiple entry points for user input are identified around lines 420-472 in
AgentServlet.java
Key Security Implications:
1. The servlet appears to be handling user input and generating HTML responses, which
is a classic XSS attack vector
2. Despite having an EscapeUtil class, some user input may be bypassing proper
sanitization
3. The LogHandler class is potentially vulnerable to log injection/log forging
attacks if it's writing user input directly to logs
Most Interesting Results:
‘‘‘
1. AgentServlet.java:472 - Direct user input handling
   - Potential unsanitized data flow from user input to output
2. AgentServlet.java:430 - Multiple vulnerability points
   - Two separate data flows identified
   - Related to parameter handling at lines 430-442
3. EscapeUtil.java:71 - Possible sanitization bypass
   - Despite being in an escape utility, this could indicate incomplete sanitization
4. LogHandler.java:82 - Log injection vulnerability
   - User input potentially flowing directly to logging functions
5. IoUtil.java:36 - I/O related vulnerability
   - User input potentially flowing into I/O operations
‘‘‘
Recommendations for Further Investigation:
1. Review the sanitization logic in EscapeUtil.java
2. Audit all HTML response generation in AgentServlet.java
3. Verify the logging implementation in LogHandler.java
4. Check if IoUtil.java properly handles user input before I/O operations
This analysis suggests that the application might benefit from implementing
consistent input validation and output encoding across all user input handling points.
```

Figure A6. Summary of CodeQL Query Results by the summarizer agent.