Evaluating Local LLMs for Vulnerability Detection and Reasoning

Riccardo Miele
dept. of mathematics, School of science
University of Padua
Padua, Italy
riccardo.miele@studenti.unipd.it

Michele Gusella
dept. of mathematics, School of science
University of Padua
Padua, Italy
michele.gusella@studenti.unipd.it

***Abstract*—In the past years Large Language Models (LLMs) have become very widespread across the world and now are used in many fields, showing great potentials, including areas like software security. In this paper, we explore the use of LLMs for identifying security vulnerabilities, focusing on smaller models. The aim of our study is to understand if small models can achieve good results in vulnerabilities detection comparing them with larger ones. Our analysis takes into account not only the efficiency and the performances of vulnerabilities detection but evaluates also the reasoning of LLMs behind to their answers. To conduct our experiment, we have used SecLLMHolmes [8], a generalized and automated framework that proposes a way to perform analysis of LLMs in the vulnerabilities' detection and in the evaluation of reasonings. For our analysis, we have adapted the framework to properly work with Ollama, a platform for running LLMs locally. Using Ollama, we have tested llama3.2, a Meta AI Large Language Model, with 3 billions parameters. We have analyzed how this small model performs with respect to the results achieved by ChatGPT 4, whose results are contained in the framework's benchmark. We have decided to focus on the eight most common vulnerabilities, such as CWE-89 (SQL Injection) and CWE-787 (Buffer Overflow).**

Our findings show that Llama3.2 can detect some vulnerabilities, in particular easy examples, but it is less accurate and consistent than larger models. The majority of the code analyzed by Llama3.2 has been flagged as vulnerable even if it was not. This behavior can be explained as the presence of some bias in the model. Furthermore, the performance are very poor if compared to greater models.

However, the performances with easy examples seem promising so a fine-tuned model with optimized dataset for vulnerability detection, may result in an useful tool to be implemented in local environment to help developers in building safe code.

***Index Terms*—Software Security, Large Language Models, Vulnerability Detection, Ollama, Llama3.2**

I. Introduction

The detection of vulnerabilities in software systems is a critical task in the field of cybersecurity. Traditional vulnerability detection methods are highly dependent on static and dynamic analysis, which can be time-consuming and error-prone. Recently, Large Language Models (LLMs), due to their easy availability, use and utility, have gained a lot of popularity. In fact, ChatGPT reached 250M of active users per week in September 2024, up from the 57M of January 2023 [1]. This extreme popularity, combined with LLMs' ability to understand and generate human-like test, has helped the exploration of the possible uses of LLMs in many fields, like software coding, due to their ability of generating code snippets [2]. Generating code using LLMs can speed up the development process by helping developers create complex code. However, using LLMs for coding also presents challenges, such as hallucinations, where the generated code looks correct but contains mistakes or hides some bugs, and non-determinism, where the same input can produce different outputs [3]. These problems make it harder to ensure reliability and consistency, requiring methods to check the validity of the generated code to reduce errors.

LLMs are also used in hardware design and verification, where they help generate code, fix errors, and find security issues. While LLMs improve efficiency, they can also introduce risks, such security mistakes in verification, requiring strong checks to ensure reliable and safe results [4].

In the last few years, Large Language Models have started also to be tested for the detection of code vulnerabilities. While LLMs have shown good potential in this area, they still struggle with issues like high rate of false positives, often flagging non-vulnerable code as vulnerable. This happens because vulnerable code and vulnerable code are quite similar because they present only small differences. However, LLMs are very good at identifying common patterns in code, effectively spotting patterns that may be involved in vulnerabilities [5]. Another study [6], showed that the GPT-3.5-Turbo model can be useful for finding vulnerabilities in software code. Although it still has some limitations, such as focusing more on certain types of vulnerabilities, it has proven to be a great tool for improving software security. It could become an important part of future systems for detecting vulnerabilities, especially when models are trained on code-based data from GitHub, rather than plain text-based data.

Large language models (LLMs) have also proven to be very useful in static code analysis. They have demonstrated improvements in detecting certain types of errors, such as variable misuse, with good accuracy rates. However, their high resource intensity has prevented their implementation in analyzers [7].

In 2024, S. Ullah et al. [8], introduced a fully automated and scalable framework, SecLLMHolmes [9] for the analysis and evaluation of LLMs in the detection of code vulnerabilities and

in reasoning about them. However, this paper highlights how these models can be used for finding code vulnerabilities but, due to the non-deterministic nature of LLMs, the evaluations of these models are sometimes not consistent. Furthermore, when the vulnerability is detected, the reasoning behind it is often inconsistent. This challenge, along to the problem of hallucinations, represents the biggest limitations of the use of LLMs for code vulnerability detection.

Starting from this framework, we have adapted it to explore how smaller LLMs (i.e. models with fewer parameters) perform in detecting and reasoning about code vulnerabilities. The adaptation was needed to make the framework properly works with Ollama, an open-source tool for running LLMs on your local machine.

The model that we have tested is Llama3.2, a Meta AI LLM with 3 billions of parameters that is very small compared to the size of bigger models, which have hundreds of billions of parameters. The purpose of our study was to evaluate the code vulnerability detection capabilities of smaller models in comparison to larger ones. Our study has showed that Llama 3.2, a small model if compared to greater ones like ChatGPT 4 which has $1.8T$ parameters, perform very poorly in vulnerability detection and reasoning. However, Llama 3.2 has showed to be able to find simple vulnerabilities, even if it's not able to see that their patched version is safe. In fact, our tested model, has flagged the majority of non-vulnerable code as vulnerable. Despite these problems, the performance with simple examples show promising results and with a fine-tuned dataset the overall performance may increase. If this can be reached, small LLMs can become a valid help to developers in building safe systems

Contributions. In this paper, we have showed that also smaller LLMs can be used for vulnerability detection. In particular:

1) We have integrated the SecLLMHolmes framework to properly work with Ollama providing a way to perform analysis of local LLMs
2) The chosen model, Llama3.2 which has only 3 billions parameters has achieved an accuracy of over 50% in vulnerability detection for simple examples

Organization. The rest of the paper is organized as follows: Section 2 presents and explains Large Language Models and the main characteristics of SecLLMHolmes framework. Section 3 presents the setup of the experiment performed. In section 4 we present the results of our experiment, and section 5 concludes the paper.

II. Background

A. Large Language Model

Large Language Models are AI systems designed to process and generate language by predicting the next token in a sequence based on prior context. This ability comes from the probabilistic modeling, where the most likely next token is chosen from the model's vocabulary. LLMs are trained on massive datasets of text and code, learning patterns and semantic relationships between words. This behavior allows the system to be very successful in tasks like text summarization, translation, question answering, and programming-related tasks such as code generation and analysis. Being chat-based, the model can be interrogated with different prompts and there are different techniques. The most simple one is Zero-Shot (ZS) scenario, where users ask the model to perform a task that the model might not have previously seen. Few-Shot (FS), instead, provides the models with few examples of inputs and expected outputs to the model before starting with actual conversation. Task-Oriented (TO) is a different type because the prompt explicitly defines a task to impose the model a specific way to respond. Role-Oriented (RO) is similar to Task-Oriented but explicitly assign a role to the model; in this way the model will answer by following the role assigned. Some role examples can be "helpful assistant" or "security expert". The particular innovation that permits LLMs to properly answer to a question is the use of the attention mechanisms, as used in Transformers. By using this mechanism, the model focuses on relevant parts of the input and this helps the understanding of the context. Another improvement of LLMs' abilities comes from the use of reinforcement learning from human feedback (RLHF). This helps the model to fine-tune and to be more human when answering.

LLMs have showed a big applicability to different fields and have also been applied to security challenges such as detecting software vulnerabilities. For these challenges, the traditional tools, like static and runtime analysis methods, required a big manual effort and often was impossible to find a way to automatize them. For this reason, LLMs can reduce the load required for detecting vulnerabilities and improve overall security all over the world, automatizing the process.

LLMs are very complex system that depends on various parameters. These can highly influence the output and the behavior of the model. The most important are model size, which refers to the number of parameters, context length, which determines how much input the model considers at once, and temperature. Temperature is the parameter that influences the randomness during the generation of the answer. In fact, lower values of this parameter make the model more deterministic by imposing the model to select the highest-probable tokens. In this way, the answers generated by the model are consistent if same question is provided different times. With higher values, the model becomes more creative and, when generating the answer, will choose the less probable token. Although the creativity can be a weakness for LLMs, it can help the model to properly answer to a bigger sets of questions. For this reason, the default value of temperature is always close to 0 but not equal.

However, LLMs are also prone to errors, like not understanding the question, or other problems like hallucinations. Hallucinations are one of the biggest or the biggest problem for LLMs, this arises when the generated answer seems correct grammatically and reasonable but, in reality, it's not. In fact, in this case, the answer is incorrect and presents some presenting some nonsensical or misleading information. This comes from

the fact that LLMs does not truly understand the question that is provided by the user, and they generate the answer by using statistical pattern learned during training. The problem comes from training set not enough big to contain all the information needed to answer. Hallucinations are even more dangerous when coming from a model that has been fine-tuned to a precise topic, because an user can expect to have received a proper answer to the provided question but this is not true.

B. SecLLMHolmes

SecLLMHolmes [9] is a generalized, fully automated, and scalable framework designed to evaluate the capabilities of Large Language Models in identifying code vulnerabilities. The primary goal of this framework is to provide a standardized approach to analyze the accuracy, reasoning capabilities, and robustness of LLMs when used as security assistants.

*1) **Framework Features**:* SecLLMHolmes evaluates LLM performance across eight key dimensions:

- Deterministic Response: Analyzing the consistency of generated responses when same question iterated multiple times.
- Performance Over Range of Parameters: Analyzing the performance of the model by varying parameters like temperature.
- Diversity of Prompts: Testing how different prompts techniques influences the model's ability to correctly respond to the provided question. These prompts are structured in different ways depending on the type of task.
- Faithful Reasoning: Measuring the correctness and coherence of the model's reasoning to understand if the generated answer is valid or not.
- Evaluation Over Variety of Vulnerabilities: Testing different types of vulnerabilities coming from real scenarios or well-known weaknesses.
- Assessment of Various Code Difficulty Levels: Evaluating performance of the model with increasing code complexity.
- Robustness to Code Augmentations: Testing robustness against trivial modifications like changes in spacing or function and variable renaming
- Use in Real-World Projects: Evaluating the ability to identify vulnerabilities in real-world software projects.

*2) **Dataset Used**:* The framework contains a dataset of three main categories:

- Hand-Crafted Scenarios: Scenarios specifically designed to test common known vulnerabilities.
- Real-World Code: Code snippets coming from real-world software projects.
- Code Augmentations: Modified code scenarios with addition of trivial or non-trivial changes.

The dataset is composed of 48 hand-crafted, 30 real-world, and 150 augmented code scenarios. Each directory contains both the vulnerable and non-vulnerable files, and the ground truth file that provides the explanation corresponding to whether the code contains a vulnerability or not. The examples of hand-crafted scenarios come from eight critical weaknesses contained in Common Weakness Enumeration (CWE), as can be seen in Table I). For instance, these weaknesses contain buffer overflows (CWE-787) or SQL injection (CWE-89). Each CWE contains six scenarios: three vulnerable cases and their corresponding patched versions. These scenarios follow a increasing difficulty structure to evaluate LLMs with different complexity levels:

- Easy: Simple programs which contains few functions and are very short. These files are named `1.c` or `1.py`, depending on the programming language.
- Medium: More complex programs in which are used external libraries for example and are longer. Also these files follow the same name formatting and are `2.c` or `2.py`.
- Hard: Complex programs with use of many functions and complicated construction. These files are `3.c` or `3.py`.

The real-world dataset, instead, contains 30 code scenarios, coming from 15 vulnerabilities from Common Vulnerabilities and Exposures (CVE). These scenarios were founded in four open-source projects. To ensure that training data of LLMs does not contain these CVEs, they are all been chosen from the year 2023 that is after the creation of the training datasets of the analyzed models. Some modifications were made to reduce the size of the programs to be able to provide to a LLM.

The last dataset, which contains code augmentations, include 150 transformed scenarios, designed to assess LLM robustness. These augmentations comes from two categories:

- Trivial augmentations: these modifications introduce syntactic noise, such as whitespace changes, renaming variables, and adding redundant statements.
- Non-trivial augmentations: these changes are more complex and modify the content keeping the vulnerability if any.

Together, these three datasets provide a comprehensive benchmark to evaluate LLM performance in detecting vulnerabilities across different levels of complexity, real-world scenarios, and robustness against modifications.

*3) **Prompts Used**:* SecLLMHolmes uses a variety of prompts to evaluate the model's capability in identifying vulnerabilities. These prompts are divided into three main categories:

TABLE I
COMMON WEAKNESS ENUMERATIONS (CWES) WITH ASSOCIATED LANGUAGES

CWE ID	Name	Language
CWE-22	Path Traversal	C
CWE-77	Command Injection	C
CWE-79	Cross-Site Scripting	Python
CWE-89	SQL Injection	Python
CWE-190	Integer Overflow or Wraparound	C
CWE-416	Use After Free	C
CWE-476	NULL Pointer Dereference	C
CWE-787	Out-of-bounds Write	C

- Standard Prompts (S): These prompts directly pose a question about a specific CWE in the given code snippet. The question can have some additional context or can assign to the LLM a predefined role, like a security expert. Another variation can be the use of few-shot examples passing to the model vulnerable and patched code to increase the model knowledge.
- Step-by-Step Reasoning-based Prompts (R): These prompts are designed to guide the model to adopt a structured approach to vulnerability detection. In fact, these prompts explicitly guide the model to divide the analysis into multiple logical steps.
- Definition-based Prompts (D): These prompts provide the model with additional background information before asking to identify the presence of a vulnerability. An example can be to pass to the model the official MITRE definition of a vulnerability; by doing so the model will have an additional security knowledge before answering.

Table II presents the 17 different prompts used in the study.

For example, Standard Prompts include variations like asking the model a direct question about a vulnerability (S1) or assigning it a role such as a 'security expert' (S3). Few-shot versions (S5, S6) provide the model with examples of vulnerable and patched code alongside reasoning. In all cases, the question remained consistent: "*Does this code contain instances of the security vulnerability classified as* CWE-XX*?*", where CWE-XX refers to the specific vulnerability being analyzed in the test.

Step-by-Step Reasoning-based Prompts (R) include techniques such as explicitly instructing the model to "think step by step" (R1) or simulating a multi-step human-like vulnerability detection process (R3). Few-shot variants (R4–R6) add real-world examples to boost the reasoning process.

Definition-based Prompts (D) introduce CWE definitions into the prompt (D1, D3) to evaluate if explicit information improve the possibility of detection. Some prompts, as the other categories, uses the few-shot reasoning and the definitions (D4, D5) to evaluate the combined effect of the these two characteristics.

4) ***How the Framework Works****:* The SecLLMHolmes framework is designed to run a series of experiments on LLMs, including determinism, parameter range exploration, and experiments with different prompting techniques, as well as code augmentations and real-world CVE scenarios. Each experiment analyzes a pre-determined set of vulnerabilities, prompts, and temperatures based on the type of experiment being conducted. The experiments in the framework are:

- Determinism Experiment aims to verify the consistency of the answers of the model when run k times (in this case $k = 10$). The prompts tested are the Standard ones, and the only files analyzed are from the CWE-89 and CWE-787. This experiment is run with two different temperatures, 0 and 0.2.
- Range of Parameters Experiment aims to analyze how the the model's answers change with increasing values of temperature. This experiment tests one prompt (S4), two CWEs (CWE-89 and CWE-787) and six values of temperature, ranging from 0.0 to 1.0.
- Prompts Experiment aims to analyze the model's performance to find the best prompt. The experiment tests all the files from the hand-crafted set (i.e., from the eight CWEs), all 17 prompts, and one temperature value.
- Code Augmentations Experiment aims to analyze the robustness of the model using trivial and non-trivial modifications.
- Real Scenarios Experiment aims to explore whether the model is able to detect vulnerabilities from a set of CVEs.

One run of the framework analyzes, in order, all the experiments explained in the list above. Once an experiment is selected, the framework begins with selecting a code file, followed by the prompt type to analyze. The selected prompt can be standard, step-by-step reasoning, or definition based. If the prompt involves few-shot examples, the system adds these examples to the system prompt before interrogating the model.

The framework then proceeds to generate predictions using

TABLE II
TYPES OF PROMPTS USED IN SECLLMHOLMES, TABLE FROM [8]

ID	Type	Description
S1	ZS	Code snippet is added to the input prompt with a question about a specific CWE (e.g., out-of-bound write, path traversal).
S2	ZS	Same as S1, but the LLM is assigned the role of a 'helpful assistant'.
S3	ZS	Similar to S1, with the LLM acting as a 'security expert'.
S4	ZS	The LLM is defined as a 'security expert' who analyzes a specified security vulnerability, without the question being added to the input prompt.
S5	FS	Similar to S1, but includes a vulnerable example, its patch, and standard reasoning from the same CWE.
S6	FS	Like S4, but also includes a vulnerable example, its patch, and standard reasoning from the same CWE.
R1	ZS	Similar to S1, but begins with "Let's think step by step" to encourage a methodical approach.
R2	ZS	The LLM plays the role of a security expert with a multi-step approach to vulnerability detection, following a chain-of-thought reasoning.
R3	ZS	A multi-round conversation with the LLM, starting with a code snippet and progressively analyzing sub-components for a security vulnerability like human security-experts.
R4	FS	Similar to S6, but the reasoning for answers involves step-by-step analysis developed by the first author.
R5	FS	Like R2, but includes few-shot examples (from the same CWE) with step-by-step reasoning for detecting vulnerabilities.
R6	FS	Similar to R5, but does not assign a specific role to the LLM in the system prompt.
D1	ZS	Adds the definition of a security vulnerability to the input prompt, followed by a related question.
D2	ZS	The LLM is a security expert analyzing code for a specific vulnerability, with the vulnerability's definition included.
D3	FS	Similar to S6, but includes the definition of the security vulnerability in the system prompt.
D4	FS	Like R4, with the addition of the security vulnerability's definition in the system prompt.
D5	FS	Similar to D4, but does not assign a specific role to the LLM in the system prompt.

the model. The question asked is always in boolean form, so the model must answer starting with "*Yes*" or "*No*". Then the model's response is evaluated based on a ground truth to determine its accuracy.

The evaluation is done by ChatGPT-4, which assesses the alignment of the model's answer with the correct answer, which comes from the ground truth directory. The framework stores the results of each experiment in a structured JSON file. This file includes several key elements:

- *content*: The model's response to the test prompt.
- *pred*: The extracted prediction, which is a boolean in the form "Yes" or "No". This depends on whether the model believes the code analyzed contains a vulnerability.
- *gpt_eval*: An evaluation boolean, in the same form as *pred*, indicating whether the model's response aligns with the ground truth.

Additionally, the framework is designed to be resilient to interruptions. If execution is stopped, the current state is saved, and upon restarting, it will automatically resume from the last experiment that was completed, ensuring continuity and efficiency in long-running evaluations.

5) ***Results****:* The framework has analyzed the following eight LLMs: chat-bison@001, codellama34b, gpt-4, codechat-bison@001, codellama7b, starchat-beta, codellama13b, and gpt-3.5-turbo-16k. Determinism Experiment, which aimed to test consistency, has showed that all the models, at the recommended temperature of 0.2, provided inconsistent responses. A temperature of 0.0, instead, improved consistency but some models, such as 'chat-bison@001' and 'gpt-4,' still present inconsistencies.

Range of Parameters Experiment, which aimed to investigate the effect of different values of temperature on LLM performance in identifying vulnerabilities, showed that increasing the temperature did not result in a general improvement in performance across models. Despite higher temperatures may have increased creativity, the experiment has found that there are no improvement in the performance with increased temperature. For this reason, the default value of the temperature in the following experiment was set to 0.0.

Prompts Experiment, which aimed to test the ability of LLMs to detect vulnerabilities in hand-crafted code scenarios testing all the prompts, showed that 'gpt-4' performed the best with a maximum accuracy of 89.5%. No single prompt has showed better performance across all models, but different models have excelled with different prompting techniques. GPT models and 'codechat-bison' performed better with step-by-step reasoning prompts, while 'chat-bison' performed best when assigned a 'security expert' role. 'Codellama34b' performed best with the $S1$ prompt, which asks if the code contains a specific vulnerability. Providing vulnerability definitions improved accuracy for 'gpt-4' and 'codellama34b', but did not have the same effect for other models.

Robustness Code Augmentations Experiment, which aimed to test performance of LLMs with trivial or non-trivial modifications, showed that trivial changes like adding whitespaces or modifying function or variable names caused LLMs to give incorrect answers and disrupt reasoning. Non-trivial augmentations also led to misdetections of vulnerabilities, with LLMs presenting biases for some library functions, flagging safe functions as vulnerable and unsafe ones as safe.

Real-World Scenarios Experiment, which aimed to test LLMs against CVEs, found that LLMs struggled to detect vulnerabilities in real-world code. In fact, the models often misidentifying patched examples as vulnerable. Few-shot prompting was ineffective while the zero-shot prompt showed improved performance, suggesting that assigning a 'security expert' role can improve results. Anyway, the poor performance suggest that we are far from be able to use these model in real-world scenarios.

III. Experimental setup

In this section, we will describe the tools used, the modifications made to the framework to meet our specific requirements, the dataset tested, and conclude with an overview of the experiment's organization.

A. Tools Used

In our study, we used several tools to test the ability of Large Language Models in code vulnerability's detection, with particular attention to small models. Our goal was not only to evaluate detection capabilities but also to explore the reasoning behind why a piece of code is vulnerable or not. To perform our experiment, we have adapted the framework SecLLMHolmes, introduced and explained in section II-B, to work with Ollama, a platform for running LLMs locally. This adaptation allowed us to install and test a local small model. The interaction between these two tools was possible due to the use of the Ollama Python library. This library provides a way to interact with a local LLM through python's snippets of code. Below, we provide a more detailed description of these tools, presenting their functionalities and relevance to our methodology:

- **SecLLMHolmes** SecLLMHolmes is a GitHub project that provides a framework and adapter for using LLMs to detect vulnerabilities in software code.
- **Ollama** is a platform designed specifically for running large language models (LLMs) locally on personal machines [10]. Unlike traditional cloud-based services, Ollama provides researchers and developers the ability to run, fine-tune, and customize LLMs directly on their own systems. This offers several advantages, including more control over the models, more flexibility in running experiments with different configurations, and the ability to preserve personal data not depending on external cloud services.

  Using this tool, we have reduced potential environment issues, eliminating network dependencies to ensure LLM to work offline. This avoids any problems due to instable network connectivity.
- **Ollama-Python library** is an open-source Python package that allows to have a Python project working with the Ollama platform [11]. It allows developers to interact

with models downloaded through Ollama and integrate them into their applications. The library provides a simple interface for invoking LLMs, sending prompts, and receiving outputs. Furthermore, the Ollama Python library's API is designed around the Ollama REST API making the use of the library easy.
This library was used to adapt the framework to work properly with our local model.

B. Framework's Adaptation

To test the framework with the chosen model, we made several modifications to adapt it to our needs.

The first modification was the implementation of an adapter, a class used to connect the LLM that we want to analyze with the framework. The adapter contains three functions:

- *prepare_prompt*: this function defines best prompting practices and rules that are specific for the LLM analyzed
- *prepare*: this function is used to define, prepare, or load the model in order to make it running
- *chat*: this function is used to pass the prompt define message structure and chat inference method

With the implementation of the adapter, we managed to connect Ollama with the framework. In this way, we added the possibility to test any LLMs coming from the platform.

The second modification made was the introduction of two input variables: *ignore_cwe* and *ignore_prompt*. These variables are passed to *prepare* function in order to ignore, respectively, some CWEs (Common Weakness Enumerations) or some prompts. This was done to limit the number of CWEs and the number of prompts analyzed by the framework, reducing the time required for a run.

The final modification was changing the model used to extract predictions and evaluate the reasoning behind the answers generated by the analyzed LLM. As default, the model used for this part was ChatGPT 4 called by the invocations of OpenAI APIs. This change was necessary because as for today, ChatGPT APIs don't have a free trial so you would need to pay for them depending on the daily usage. To solve this problem, we have created a new function *call_structured_ollama* to create a new local model, using Ollama, for the extraction of the predictions and the evaluation of the reasoning.

To perform our task, we also have excluded the computation of rogue and cosine similarity from our tests and we have commented all the calling to functions related to CVEs and Code Augmentations.

C. Experimental Dataset

In our experiment, we have decided to limit our evaluation only to the CWEs contained in the *hand-crafted* dateset of the framework. This has reduced the number of vulnerable or non-vulnerable examples to six elements for each CWE, with three vulnerable and three not vulnerable. Table I provides the list of all CWEs analyzed and the type of programming language associated with each one. The reason behind this choice is to focus firstly on these common type of vulnerabilities, leaving for a future work the possibility to analyze other types.

D. Experiment

Our experiment wants to explore how small LLMs perform in vulnerability detection and reasoning. The LLM model chosen for our test is LLama3.2 [12], a Meta AI LLM with 3 billions parameters. This model is also used in the part of evaluation of the local model, extracting predictions and reasoning.

The CWEs analyzed are from the hand-crafted dataset, as mentioned previously. Some examples are Out-of-bounds Write (CWE-787) or SQL Injection (CWE-89). The prompts used are all the ones contained in the framework so promptSi with $i \in [1, 6]$, promptDj with $j \in [1, 5]$ and promptRk with $k \in [1, 6]$. We have performed three experiments, coming from the original framework, which are:

- Determinism: to test the consistency of the generation of the answer with two temperatures: 0.0 and 0.2. Every test is done five times and these tests are limited to CWE-787 and CWE-89.
- Range Parameters: to test how, keeping the prompt fixed to PromptS4, the temperature influences the answers generated. Every test is done five times and these tests are limited to CWE-787 and CWE-89.
- Prompts: to generate an answer for each CWE and for each prompt.

Summing up, we have done: 240 tests for determinism, 120 tests for range parameters and 816 tests for prompts, for a total of 1176 tests.

After this initial part, where we have tested the chosen model, Llama3.2, on all the three experiments, we have obtained an overview of the capabilities of the model. Starting from these results, we managed to extract the two best prompts, one for the Zero-Shot category and one for Few-Shots, and the best temperature. The two prompts have been extracted from the Prompts experiment while the temperature was found from Range Parameters.

With this new configuration, we have run two times the Prompts Experiment with an additional third time when there were some inconsistency between the two initial runs.

IV. Analysis and Results

All the experiments were performed with two different computers: a HP Probook laptop with Intel i5 10th generation, integrated GPU, 16GB RAM and Ubuntu OS, and a Desktop PC with Intel i5 12th generation, AMD Radeon RX 6650 XT GPU, 16GB RAM and Windows 11 OS.

The objective is to compare the evaluated model that we tested, Llama3.2, with GPT-4 whose results are published by the researchers on their github.

A. Evaluation of results

When evaluating the responses, there were multiple metrics that we could have chosen, one of which is the F1-score computed as follows:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

where precision and recall are defined as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

A true positive would be a correct prediction of a vulnerability, a true negative would be a correct prediction of a file that is not vulnerable, a false positive would be predicting a vulnerability on a non-vulnerable file and a false negative would be not predicting a vulnerability when there is one.

However, there was a major issue with this approach. The F1-score did not take into account whether the evaluator model's evaluation (*gpt_eval*) was correct or not. This meant that even if the model's prediction was correct but the reasoning behind it was incorrect (i.e., *gpt_eval* was "No"), the F1-score would still count it as correct, leading to misleading results. To address this problem, we changed metric choosing the same one used in the reference paper which is the accuracy, defined as the ratio of correct predictions to the total number of predictions ensuring that only answers with both correct prediction and correct reasoning were considered valid. The formula we used is:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

All the accuracies presented in the following tables are expressed as percentages, meaning that the value is calculated as:

$$\text{Accuracy } \% = \text{Accuracy} \times 100$$

B. Preliminary determinism experiment

We ran the determinism experiment to check the consistency of Llama3.2 responses. For this experiment, we set k to 5, meaning we ran the model 5 times with the same prompts and files. We found that, in general, the responses were consistent on all runs. However, on a few occasions, one response would differ from the others. For example, if we expected 5 identical responses, 4 of them were consistent, while one response would be different. When this anomaly occurred, it was limited to just one iteration, and the other results remained unchanged.

C. First experiment: all prompts and all temperatures

To find the best Zero-shot prompt and Few-shot prompt for each LLM model, the first experiment runs the evaluated model on the entire hand-crafted dataset using all prompts shown on Table II. The results indicate that the best prompts for Llama 3.2 are promptR5 (Few-shot) and promptD2 (Zero-shot), achieving an accuracy of 27% on promptR5 and 23% on promptD2 whereas the best prompts for GPT-4 are promptR2 (Zero-shot) and promptR6 (Few-shot) achieving 89.5% accuracy on both of them. This initial experiment already highlights the superiority of GPT-4 over Llama3.2, suggesting that GPT is much more reliable in identifying vulnerabilities. These results are shown in Table III. Then using these results we can also find the best temperature by computing the accuracy score in the range parameters experiment mentioned in Section II-B4, which tries different values of temperature. The best temperature for Llama3.2 is 0.0 with an accuracy of 25% whereas the best temperature for GPT-4 is also 0.0 with an accuracy of 50%. This result aligns with the expected behavior found with other LLMs.

TABLE III
ACCURACY OF LLAMA3.2 AND GPT-4 ON THEIR BEST PROMPTS

Model	Prompt	Accuracy
Llama3.2	PromptR5	27%
Llama3.2	PromptD2	23%
GPT-4	PromptR2	89.5%
GPT-4	PromptR6	89.5%

Challenges during this run. The results of the first experiment are not 100% reliable because it was conducted over multiple days and on our two different PCs due to the high time required for a single test, which was circa three minutes. Since running the entire experiment from start to finish on a single machine would have taken too long, the process was split across different sessions and hardware. This introduces potential inconsistencies which could have affected the outcomes.

D. Second experiment: best prompts and temperature

Then, after understanding which are the best prompts for each model, the second experiment was conducted by running the evaluated model only on its best prompts, promptR5 and promptD2, and by fixing the temperature to 0.0 which is the best one. For this experiment, the results were obtained from three separate runs: the first and the third run on the slightly better PC, the Desktop PC, and the second run on the slightly worse PC, the laptop one. Unlike the first experiment, where the runs were split across multiple days and different machines, this time each run was executed from start to finish without any interruptions, so the results are more consistent and reliable. For this experiment the time required was six hours and an half for the laptop and two hours for the Desktop PC.

The results indicate that Llama 3.2 did not perform well, achieving an average accuracy of 17% on promptR5 and 14.5% on promptD2. The overall results are shown in Table IV and Table V. While the results indicate that the LLM performed poorly overall, there are two main points to notice: it showed a better ability to classify vulnerable files compared to non-vulnerable ones and it performed better on easier files (whose number in the name was lower) with respect to files that had more difficult vulnerabilities to find.

The best accuracy for vulnerable files reached 31.91%, while almost no non-vulnerable files were correctly classified. This discrepancy could be due to a bias in the model's training data since vulnerabilities often have more distinctive patterns such as specific function calls, insecure API usage, or well-known security flaws, so the model might have learned to recognize these patterns more effectively. On the other hand,

non-vulnerable code is generally similar to vulnerable code but with small changes, making it harder for the model to distinguish it from actual vulnerable code. This can be explained by the fact that LLMs work with tokens rather than words. A line or a set of lines in the code can be encoded in a specific way, and even if something is changed within that code, the LLM might perceive it as exactly the same. Additionally, if the dataset used for fine-tuning the model contained more examples of vulnerabilities than safe code, the model could have developed a bias to predict more often the presence of vulnerabilities.

The other trend in the results is that the LLM performed better on files with lower numbers, such as *1.c* and *1.py*, compared to files with higher numbers in the name. The model tends to perform better at identifying vulnerabilities in easier files primarily due to the simpler structure and logic of the code. In such files, the code is usually more straightforward, with fewer dependencies and less complex data manipulation. This makes it easier for the model to understand the flow of the program and spot potential security risks. For example, common vulnerabilities like buffer overflows or improper error handling often arise in predictable coding patterns, such as the use of unsafe functions like *strcpy()* or *fopen()*. Since these patterns are more likely to appear in simpler code, the model can quickly recognize them and flag potential issues. Additionally, files with simpler code typically do not rely on external factors like environment variables or external libraries, which can add another layer of unpredictability. Without these dependencies, the model can focus purely on the code itself, allowing for more accurate identification of vulnerabilities.

Challenges in Evaluation. An important factor that may have affected the results is the shift in the framework used for extraction and evaluation. Initially, ChatGPT was responsible for both extracting the response from the evaluated model and assessing its correctness comparing the model's response with the ground truth. However, we changed the evaluator model to Llama3.2. This introduced a new source of error, where even when the evaluated model (Llama 3.2) provided a correct response with a valid explanation, the extraction or evaluation process could misinterpret it. As a result, some correct answers were wrongly classified as incorrect, making the performance of the evaluated model seem worse than it actually was. That said, while these errors do exist, they are negligible as they do not occur frequently. The overall trend in the results remains valid, and the occasional misclassifications do not significantly impact the conclusions drawn from the experiment.

E. GPT-4 comparison

Table VI shows the results achieved by GPT-4. Other than the fact that Llama's performances were taken using normal PC hardware, it is important to highlight that GPT-4 is a large model with an estimated amount of 1.8 trillion parameters [13], whereas Llama with only 3 billion parameters is considerably smaller.

The first observation is that there are significant differences in performance between the two models. Llama3.2 model generally struggled with accuracy, particularly when classifying non-vulnerable files. On the other hand, GPT-4 showed higher accuracy across most files, especially in classifying vulnerable files, with a score of 81.89%. While the Llama3.2 model had low accuracy in both vulnerable and non-vulnerable classifications, GPT-4 demonstrated a more balanced performance, even if it wasn't flawless.

TABLE IV
SORTED ACCURACY RESULTS ACROSS THREE RUNS OF LLAMA3.2

File	Run 1	Run 2	Run 3
1.c	33.33%	58.33%	58.33%
1.py	50%	50%	25%
3.py	0%	50%	0%
3.c	33.33%	16.67%	33.33%
2.py	0%	0%	25%
2.c	0%	18.18%	8.33%
p_3.c	0%	8.33%	0%
p_1.c	0%	0%	0%
p_2.c	0%	0%	0%
p_1.py	0%	0%	0%
p_2.py	0%	0%	0%
p_3.py	0%	0%	0%

TABLE V
CATEGORY-WISE ACCURACY RESULTS OF LLAMA3.2

Category	Run 1	Run 2	Run 3
Vulnerable	20.83%	31.91%	29.17%
Non-Vulnerable	0%	2.17%	0%

By looking at the generated responses, three main differences can be found.

1) Overstimation of risk: We observed that Llama3.2 often misclassifies non-vulnerable code as vulnerable, whereas GPT-4 generally provides more accurate classifications. This indicates that Llama3.2 has a tendency to overestimate risk. For example: in CWE-190 (Integer Overflow), Llama3.2 flagged vulnerabilities even in cases where the code already had safeguards in place, such as checking for overflow conditions before performing arithmetic operations. On the other hand, GPT-4 correctly recognized that the safeguards were sufficient and did not label those cases as vulnerabilities. Similarly, in CWE-22 (Path Traversal), Llama3.2 incorrectly identified safe implementations as vulnerable, failing to account for built-in security mechanisms like *realpath()* and input sanitization. GPT-4, in contrast, acknowledged these mitigations and avoided false positives.

2) Misinterpretation of Context: Llama3.2 sometimes misunderstood the intention of the code: in multiple cases, it flagged vulnerabilities that were not actually exploitable. It sometimes misread function logic, failing to consider how different parts of the code interact. GPT-4 was more effective at understanding data flow and understanding complex program structures. Example: In an integer overflow vulnerability case, the code included explicit bounds checks before performing arithmetic operations. Llama3.2 still marked it as vulnerable, missing the meaning of the safeguard. GPT-4 correctly identified and explained why the existing checks were sufficient to prevent overflow.

TABLE VI
GPT-4 MODEL RESULTS SORTED BY ACCURACY

File	Accuracy
p_1.py	100%
p_2.py	94.12%
3.c	92.16%
3.py	91.18%
1.c	84%
1.py	82.35%
2.py	75%
p_1.c	74.26%
p_2.c	72.55%
2.c	68.32%
p_3.c	51.96%
p_3.py	47.06%
Vulnerable	81.89%
Non-Vulnerable	69.78%

3) Real-World Scenarios: GPT-4 demonstrated a better understanding of practical security issues, such as identifying real-world attack vectors rather than just theoretical risks, recognizing when a vulnerability was unlikely to be exploited due to practical constraints or suggesting more precise fixes that balanced security and performance. For example, in path traversal vulnerabilities, Llama3.2 suggested mitigations that were too simple like banning certain characters in file paths. On the other hand GPT-4 recommended a more robust approach, such as using *realpath()* and using directory whitelisting.

In a production environment, given the higher accuracy and more consistent results from GPT-4, it would be the preferred model for tasks that require reliable classification, such as vulnerability detection. GPT-4 is much better at identifying vulnerable files accurately, making it more useful for real-world applications. However, Llama could still be used in situations where there are fewer computing resources available. Still, for important tasks that require high reliability, GPT-4 would be the better option because of its stronger performance.

V. CONCLUSION

Our study looked at how well smaller LLMs can detect vulnerabilities and explain their reasoning, comparing it with bigger LLM models. We specifically compared Llama 3.2 with a larger model, GPT-4 using the SecLLMHolmes framework to automate the process of collecting responses. We found that while smaller models can detect some vulnerabilities, they are much less accurate and consistent in their reasoning compared to larger models. Llama 3.2 tends to classify code as vulnerable more often than it should, showing a possible bias in its detection. Additionally, its performance dropped when dealing with more complex vulnerabilities, showing that smaller models struggle more when analyzing complicated code structures.

Despite these challenges, the study also demonstrates that smaller models can still provide useful insights in specific contexts, particularly when computational resources are constrained. With further advancements in model training techniques and dataset optimization, the gap between small and large LLMs could be reduced, resulting in improved accuracy, better reasoning capabilities, and more efficient vulnerability detection even on resource-constrained hardware. This also means that such models could be run locally, for example, within an IDE, to automatically detect vulnerable code as the programmer is writing it, enhancing security during development.

REFERENCES

[1] David Curry, “ChatGPT Revenue and Usage Statistics (2024)“, November 2024, https://www.businessofapps.com/data/chatgpt-statistics/

[2] H. Li and L. Shan, ”LLM-based Vulnerability Detection” 2023 International Conference on Human-Centered Cognitive Systems (HCCS), Cardiff, United Kingdom, 2023, pp. 1-4, doi: 10.1109/HCCS59561.2023.10452613.

[3] A. Fan et al., ”Large Language Models for Software Engineering: Survey and Open Problems,” 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), Melbourne, Australia, 2023, pp. 31-53, doi: 10.1109/ICSE-FoSE59343.2023.00008.

[4] R. Kande, V. Gohil, M. DeLorenzo, C. Chen and J. Rajendran, ”LLMs for Hardware Security: Boon or Bane?,” 2024 IEEE 42nd VLSI Test Symposium (VTS), Tempe, AZ, USA, 2024, pp. 1-4, doi: 10.1109/VTS60656.2024.10538871.

[5] M. D. Purba, A. Ghosh, B. J. Radford and B. Chu, ”Software Vulnerability Detection using Large Language Models,” 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), Florence, Italy, 2023, pp. 112-119, doi: 10.1109/ISSREW60843.2023.00058.

[6] V. Akuthota, R. Kasula, S. T. Sumona, M. Mohiuddin, M. T. Reza and M. M. Rahman, ”Vulnerability Detection and Monitoring Using LLM,” 2023 IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE), Thiruvananthapuram, India, 2023, pp. 309-314, doi: 10.1109/WIECON-ECE60392.2023.10456393.

[7] V. N. Ignatyev, N. V. Shimchik, D. D. Panov and A. A. Mitrofanov, ”Large language models in source code static analysis,” 2024 Ivannikov Memorial Workshop (IVMEM), Velikiy Novgorod, Russian Federation, 2024, pp. 28-35, doi: 10.1109/IVMEM63006.2024.10659715.

[8] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, “LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks,” in 2024 IEEE Symposium on Security and Privacy (SP), Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 862–880.

[9] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini. (n.d.). [Software]. GitHub. https://github.com/ai4cloudops/SecLLMHolmes. Accessed [02/01/25].

[10] Ollama. (n.d.). Ollama library [Software]. GitHub. https://github.com/ollama/ollama. Accessed [02/01/25].

[11] Ollama. (n.d.). Ollama Python library [Software]. GitHub. https://github.com/ollama/ollama-python. Accessed [02/01/25].

[12] Ollama. (n.d.). llama3.2 LLM model [Software]. Ollama. https://ollama.com/library/llama3.2. Accessed [02/01/25].

[13] Maximilian Schreiner, “GPT-4 architecture, datasets, costs and more leaked“, July 2023, https://the-decoder.com/gpt-4-architecture-datasets-costs-and-more-leaked/