Enhancing Code Security Through Open-source Large Language Models: A Comparative Study

Norah Ridley(✉), Enrico Branca, Jadyn Kimber, and Natalia Stakhanova

University of Saskatchewan, Saskatoon, Saskatchewan, Canada
norah.ridley@usask.ca

Abstract. Significant advances in the language processing field are providing new innovations, including the ability to analyze code for weaknesses. Typically, analyzing code security is performed by tools that use known vulnerable patterns, which may not adequately represent the intricacies of vulnerabilities in real-world projects. Such tools can fail to detect non-standard weaknesses in code samples, potentially leading to a loss of personal and financial information for end users of the code. Using language-based models to detect weaknesses that would have otherwise been missed by the currently available analysis tools is a promising new avenue of vulnerability detection. In this research, we employ 25 different models to evaluate the security of code samples. Using an existing dataset of insecure code, we prompt each model to detect weaknesses in the vulnerable code. Our findings indicate that most models are ill-equipped to deal with insecure code. Through our analysis, we identify strategies for improving weakness detection using language models.

Keywords: Large language model · Code weakness · Code security

1 Introduction

The wide application of large language models (LLMs) is changing our perception of technology. Significant advances in the language processing field are providing new innovations, including the ability to write, analyze, and correct code written in different programming languages. The innovations in AI are continuously increasing the capabilities of language models and allowing for a more flexible approach to code generation.

Several studies have examined the code quality produced by LLM tools, assessing correctness, validity, dependability, and maintainability [1, 2, 6, 34, 41]. While these considerations are extensively explored, the security aspect often remains overlooked. Recent research studies have explored the presence of vulnerabilities in code generated by GitHub's Copilot [29], the ability of OpenAI's Codex to generate hardware security assertions [20], and the security implications of using AI-generated code [33].

In this work, we perform a comprehensive comparative analysis of popular LLM tools to explore their ability to recognize the presence of security weaknesses in code. Typically, code security analysis is performed by tools relying

on known vulnerable patterns. Generating these patterns is an error-prone and time-consuming process. Using language-based models to detect code weaknesses can significantly expedite this process, potentially offering a promising new avenue of vulnerability detection.

In this study, we employ 25 different LLMs to evaluate the security of code samples. Using a dataset of insecure code, we ask each of our models to detect common weakness enumerations (CWEs) in the code samples. CWEs are a widely used community-developed list of software and hardware weaknesses that provide a baseline for detecting vulnerabilities in code. Through this approach, we explore how LLMs have evolved over time, looking at how models based on GPT-3 were improved to create new LLaMA models. To gain a comprehensive understanding of the current state of language models and code weakness detection, we focus our study on large models. Our smallest model is based on seven billion parameters (7B) and trained using one trillion tokens while our largest model uses 70 billion parameters and are trained using 1.4 trillion tokens.

Our work presents several findings that expand our understanding of LLMs as code analysis tools:

- Overall, our findings show that models are ill-equipped to offer sufficient reasoning for identifying code weaknesses. Models that are trained on general datasets outperform models that are specifically trained on coding data.
- Our analysis highlights that models tended to identify more CWEs that are represented as text rather than CWEs that must be inferred by the LLMs. Hence, LLMs should be trained on both code and code evaluation data to develop sufficient reasoning about code weaknesses.
- We investigate strategies for improving weakness detection. Our results suggest that pairing static analysis tools with LLMs can detect a wider range of CWEs in code.

2 Background and Related Work

Large language models overview. Large language models are transformer-based neural networks that are trained on a large amount of text datasets. The datasets often include code samples written in different programming languages. These models have billions of parameters, which are the adjustable elements that allow the model to learn the relationships and patterns in text data. Input to an LLM is known as a *prompt*, which is a limited length sequence of tokens. Large language models have a wide range of applications, including natural language understanding, text generation, language translation, and code analysis. Building an LLM is a complex and expensive task. Training a model from scratch requires a significant amount of time and resources. However, it is often easier to fine-tune a pre-existing model for a specific task. Fine-tuning through data, in the context of this work, refers to adjusting the weight of input training data to optimize outputs.

Related work. Code vulnerability analysis has been at the centre of research in computing for over a decade. Longstanding papers in the research area of vulnerability detection have often focused on identifying code weaknesses to prevent code vulnerabilities from occurring. The application of machine learning as a means of detecting code vulnerabilities has been the focus of Yamaguchi et al. [40] and Bilgin et al. [5]; both of which outline specific techniques for detecting vulnerabilities using AI-based language processing. While these works are important, our paper maintains a particular focus on using LLMs for detecting code weaknesses.

A few works have focused on the role of LLMs in creating and editing code. Taecharungroj et al. [38] explored a wider view of an LLM's ability to write code. Surameery et al. [37] used LLM models to detect bugs in code. These works explore the capabilities of LLMs, but they do not delve deeply into the weakness detection capabilities of LLMs.

Some works have explored LLMs as code vulnerability detection tools such as Khoury et al. [21], which explored the performance of GPT-3.5 in detecting vulnerabilities. Our work differs from this approach in two ways. First, we focus on code weaknesses rather than vulnerabilities. Second, we employ a more comprehensive approach by analyzing and comparing the performance of a large cross-section of LLMs. Siddiq et al. [35] provided the framework for the security evaluation of generated code. Lastly, Pearce et al. [30] recently published a work detailing a large-scale comparison of LLM-generated code. However, their study focuses on repairing vulnerabilities rather than detecting weaknesses. As such, there has never been a large-scale comparison of LLMs in the context of detecting weaknesses. Thus, we address this gap in the existing literature by evaluating the abilities of 25 popular LLMs in assessing vulnerable code.

3 Methodology

In our study, we evaluate a set of 25 widely-used LLMs. To establish a reliable baseline for our subsequent evaluation, we used an existing dataset that contained vulnerable and generally weak code, which served as a reference point for comparing the models' evaluations of code vulnerabilities. Additionally, we enhanced this evaluation by incorporating CodeQL [22], a static analysis tool.

3.1 Large Language Models

Our work focused on 25 popular open-source LLMs employed by both industry and academia. We selected LLMs with over 500 downloads in the past month[1] from the Hugging Face platform [13], an open-source AI community. The exceptions were the two Samantha models, which we included due to their specialized training datasets.

The initial version of GPT-3 that was released by OpenAI displayed three key abilities: language generation, in-context learning, and world knowledge [14].

[1] At the time of writing

Since the GPT model is proprietary, we focused our efforts on evaluating approximate open source alternatives; one such alternative to GPT-3 is LLaMA, an open-source LLM released by Meta.

Our models include the original *Guanaco 7B* [23] model, which is a fine-tuned variant of the original LLaMA model with four-bit QLoRA tuning. We included one other model based on the original LLaMA model in the form of Vicuna, a LLaMA model that is fine-tuned with user conversations from ShareGPT [19]. Additionally, we used the *Samantha 33B* model, which is based on Vicuna. This model was trained on a curated dataset of 6,000 conversations in ShareGPT [15].

We also leveraged models trained against LLaMA models such as *Wizard Vicuna 7B Uncensored* and *Wizard Vicuna 30B Uncensored*, which removes alignment and moralizing data from the original Wizard Vicuna model [16].

LLaMA 2 models are the next generation of LLaMA [26] that improve on the original LLaMA model by training on 40% more tokens, having a longer context length, and using grouped-query attention. We included the LLaMA 2 models with seven billion and 13 billion parameters in our analysis, referring to these models as *LLaMA 2 7B* and *LLaMA 2 13B*, respectively.

The *Luna AI LLaMA Uncensored model* fine-tunes the LLaMA 2 model using 40,000 long-form chat discussions from real users [39]. In addition, we included the seven billion parameter, 13 billion parameter, and 70 billion parameter versions of the StableBeluga2 [25] model in our analysis, consequently referred to as *StableBeluga2 7B*, *StableBeluga2 13B*, and *StableBeluga2 70B*. StableBeluga2 models are trained on an Orca style dataset [27] and fine-tuned on LLaMA2. Our evaluation also includes the *LLaMA 2 Coder 7B* model, which was based on the LLaMA2 model and trained on the CodeAlpaca 20K dataset (based on Stanford Alpaca) [32].

Additionally, we included the LLaMA 2 version of Guanaco, called LLaMA 2 Guanaco QLoRA. We analyzed the seven billion parameter, 13 billion parameter, and 70 billion parameter versions of this model (*LLaMA 2 7B Guanaco QLoRA*, *LLaMA 2 13B Guanaco QLoRA*, and *LLaMA 2 70B Guanaco QLoRA*).

Another model in the GPT-3 series was the initial Codex, which has the ability to understand and generate code. One approximate open-source alternative is the Salesforce GodeGen models. Thus, we included the *CodeGen2.5 7B Mono* model [12] in our evaluation.

Models in the GPT-3.5 series not only have the ability to understand and generate code, they also exhibit a capability for complex reasoning and possibly long-term dependency. The StarCoder model is an approximate open source alternative. We included the *StarCoder* and *StarCoderPlus* models in our set since both of these models are trained on over 80 programming languages [24].

The *Chronos Hermes 13B* model [4] uses a mixture of the LLaMA derived model, Chronos-13B, and the Nous Hermes model trained on a variety of data including data from technical forums and repositories, as well as coding documentations for various products.

We also evaluated the mixed model *Nous Hermes LLaMA 2 13B*, which was fine tuned on over 300,000 instructions to produce an enhanced LLaMA 2 13B model that aims to rival the performance of the GPT-3.5-turbo model [31].

We included in our analysis several LLaMA-based models that were trained using output based on GPT-4.0. Similar to GPT-3.5, GPT-4.0 was trained on technical forums, code repositories, and coding documentation. We included the *Airoboros 7B GPT4-1.2* and *Airoboros 13B GPT4-1.4* models, which are QLoRA models that were trained on, among other things, programs written in multiple programming languages, and fine-tuned on entirely synthetic data using seven billion and 13 billion parameters, respectively [3].

We evaluated the *OpenOrca Platypus2 13B* model that was trained on the Platypus dataset, which includes Python coding exercises [23]. This model mixes the LLaMA 2-based Platypus2 model with the GPT-4 based OpenOrca model [28].

Our analysis also included the *Falcon 7B Instruct* model, a causal decoder-only model [18] trained on a variety of instruct and chat datasets, and its variant *Samantha Falcon 7B* model, which like Samantha 33B, was trained on ShareGPT data.

3.2 Code dataset

To assess the models' abilities to recognize insecure code, we employed the SecurityEval dataset produced by Siddiq and Santos [35]. This dataset includes code samples collected from four different sources: CodeQL [22] documentation, the CWE website [7], from SonarSource [36] rules, and the study by Pearce et al. [29]. To supplement their dataset, the authors generated additional code samples. Each sample targets a specific CWE. The collected samples were assessed by their original source and assigned a CWE. The authors of the SecurityEval dataset used these CWE assignments to organize their dataset by CWE. The generated samples were created to address specific CWEs that were not already in the dataset, meaning that the authors intentionally developed insecure code. In the current version of this dataset, there are 121 programs written in Python language targeting 69 CWEs. Table 1 shows the distribution of code samples in the dataset sorted by CWE.

3.3 Baseline analysis

To determine how efficient the selected models are at detecting code weaknesses, we use CodeQL [22], an open source semantic code analysis engine created by the Github Security Lab. CodeQL leverages a set of predefined or customized queries to analyze code for security issues and code correctness. We scanned all 121 programs from SecurityEval dataset using the CodeQL tool. These results establish our baseline when analyzing the LLMs' assessments of the dataset, allowing us to gain an understanding of the models' performances against a widely used static analysis tool.

Table 1: CodeQL performance on the SecurityEval dataset

	Code samples															
CWE Number	CWE-020	CWE-022	CWE-078	CWE-079	CWE-080	CWE-089	CWE-090	CWE-094	CWE-095	CWE-099	CWE-113	CWE-116	CWE-117	CWE-1204	CWE-193
SecureEval	6	4	2	3	1	2	2	3	1	1	2	2	3	1	1
CodeQL	0	7	2	8	0	0	4	2	0	0	0	1	1	0	0
	CWE-215	CWE-250	CWE-252	CWE-259	CWE-269	CWE-283	CWE-285	CWE-295	CWE-306	CWE-319	CWE-321	CWE-326	CWE-327	CWE-329	CWE-330
SecureEval	1	1	2	2	1	1	1	3	1	4	3	2	4	1	2
CodeQL	1	0	0	0	0	0	0	1	0	0	0	2	6	0	0
	CWE-367	CWE-377	CWE-379	CWE-385	CWE-400	CWE-406	CWE-414	CWE-425	CWE-434	CWE-454	CWE-462	CWE-477	CWE-502	CWE-521	CWE-522
SecureEval	2	2	2	1	2	1	2	1	2	1	2	2	4	2	2
CodeQL	0	2	0	0	0	0	0	0	0	0	0	0	2	0	0
	CWE-605	CWE-611	CWE-641	CWE-643	CWE-703	CWE-730	CWE-732	CWE-759	CWE-760	CWE-776	CWE-798	CWE-827	CWE-835	CWE-841	CWE-918
SecureEval	1	6	1	2	3	3	1	1	1	1	2	1	1	1	2
CodeQL	0	2	0	1	0	3	1	0	0	1	51	0	0	0	2
	CWE-200	CWE-209	CWE-595	CWE-601	CWE-331	CWE-339	CWE-941	CWE-943	CWE-347						
SecureEval	1	1	1	5	1	2	1	1	4						
CodeQL	0	0	0	4	0	0	0	0	0						

3.4 Model assessment

Similar to the baseline analysis, we prompted each model to evaluate the 121 code samples. Since manually querying each model for each code sample takes a significant amount of time, we automated this process with the Python library, llama2-wrapper [17]. Using a custom Python script, the models evaluated the code samples using the following query: *Does the following code contain a CWE-ID? If so, what is the CWE-ID? [code sample]*. By design, each model's responses (and similarly the prompts) are limited by character count and time; once these limits are exceeded, the response is terminated and the model is presented with the next query. We query each model only once regardless of the received response quality.

4 Results

Experimental setup. We downloaded and ran the models on a Linux Pop OS 22.04 machine with two NVIDIA GeForce RTX 3090 GPUs. Each GPU has 24GB of memory, 10,496 CUDA cores, and 328 tensor cores, providing us with sufficient power to evaluate the collected models.

CodeQL assessment. Table 1 presents the results of baseline assessment with CodeQL. Although CodeQL is generally regarded as a state-of-the-art code assessment tool, our analysis shows that its results in most cases do not agree with the dataset labelling. Overall, CodeQL identified CWEs only in 50 samples from the dataset. CodeQL identified fewer CWEs than what was present according to the dataset labelling. For four different CWEs, CWE-079, CWE-90, CWE-327, and particularly with CWE-798 (the use of hard coded credentials), CodeQL detected more instances of these CWEs than were actually present.

Since there is no guarantee that initial dataset labelling is more accurate than the CodeQL assessment, we further provide comparison along both assessment results.

LLM assessment compared to the SecurityEval dataset. Table 2 shows the outcome of each model's attempt to detect CWEs compared to the SecurityEval

Table 2: LLM code assessment compared to the SecurityEval dataset labelling

Model	**Provided CWE matched**	**CWE did not match**	**No response**	**No CWE found**	**Other response**
LLaMA 2 13B	6	99	4	11	1
LLaMA 2 7B	3	79	8	22	9
StarCoderPlus	3	17	85	16	0
OpenOrca Platypus2 13B	2	24	3	92	0
Nous Hermes LLaMA 2 13B	1	11	89	20	0
LLaMA 2 13B Guanaco QLoRA	1	28	74	17	1
StarCoder	1	41	74	4	1
Vicuna 33B	0	0	121	0	0
Airoboros 13B GPT4-1.4	0	0	121	0	0
LLaMA 2 7B Guanaco QLoRA	0	24	49	44	4
StableBeluga 7B	0	10	83	28	0
CodeGen2.5 7B Mono	0	1	120	0	0
Airoboros 7B GPT4-1.2	0	0	121	0	0
Guanaco 7B	0	0	121	0	0
Falcon 7B Instruct	0	0	4	117	0
Wizard Vicuna 7B Uncensored	0	0	121	0	0
Samantha 33B	0	0	121	0	0
Luna AI LLaMA Uncensored	0	10	35	76	0
StableBeluga 13B	0	5	58	58	0
Chronos Hermes 13B	0	0	121	0	0
LLaMA 2 Coder 7B	0	0	121	0	0
Samantha Falcon 7B	-	-	-	-	-
Wizard Vicuna 30B Uncensored	-	-	-	-	-
LLaMA2 70B Guanaco QLoRA	-	-	-	-	-
StableBeluga2 70B	-	-	-	-	-

baseline. For each model, we categorize the models' responses to the 121 Python samples as follows:

- **Provided CWE matched.** The model identifed a CWE in a given code sample that matched the SecurityEval CWE labelling for that sample.
- **CWE did not match.** The model identified a CWE that did not match the CWE indicated in the dataset.
- **No response.** The model did not respond to our query.
- **No CWE found.** The model assessed the code sample and stated that there were no CWEs present in the sample.
- **Other.** Model responded with uncertainty or with a fragmented response.

As our results in Table 2 show, the vast majority of models did not agree on the presence of CWEs according to the SecurityEval labels.

Only seven models responded with at least one identification of a CWE that matched the baseline. Among them, LLaMA 2 13B appears to be the most successful model (detected six CWEs matching the SecurityEval labelling). This

means that the model most in line with SecurityEval labelling agreed with it only five per cent of the time. The StarCoderPlus model performed the second best after the LLaMA 2 models. Surprisingly, the most common result from the models was a failure to answer, which mostly consisted of the models returning a timeout error when attempting to provide a response. Of the tested models, eight returned this response for all 121 code samples.

Interestingly, most of the models that were trained or fine-tuned using technical repositories and documentation did not stand out in their performance. For example, the Airoboros 7B GPT4-1.2 model, which was trained on data specifically focused on coding examples that were written in multiple programming languages, failed to identify any CWEs in the code samples. Rather, it did not answer the prompt for any of the samples as shown in Tables 2 and 3. Similarly, the Chronos Hermes 13B and LLaMA 2 Coder 7B models gave no response in all cases.

Similar to incorrect responses, a significant number of responses from models claimed that no CWEs were present in code despite the indication of the baseline. Three models returned this response for over half the submitted code while nine models never returned this response.

Falcon 7B Instruct was an outlier in this case, claiming a lack of weaknesses in code for most (117 out of 121) of the analyzed samples. The Falcon 7B Instruct model's pre-training data consisted of public data from the web. The model was optimized, meaning that only the data with the most positive hits was retained. As a result, it appears that the Falcon 7B Instruct model can not recognize code.

Lastly we received a negligible amount of miscellaneous responses from the models which did not fit into any of the previous categories.

LLM assessment compared to the CodeQL results. Table 3 presents the results of comparing the models' assessments for the 50 code samples that were deemed insecure by the CodeQL scan. The results show the number of times a model's assessment of a specific code sample matched with results of the CodeQL scan for that file.

The LLaMA 2 13B performed the best, identifying four CWEs that agreed with CodeQL's results. Interestingly, this model also performed the best according to the original dataset labelling. However, it also had the highest number of cases (44 out of 50) identifying CWEs that did not correspond with CodeQL's assessment. The StarCoderPlus model, which was intended to help developers with coding tasks, performed the second best.

Nine of the models did not answer the query for all 50 samples of our subset. Among them were the models based on GPT-3.5 and GPT-4: Airoboros and Chronos Hermes. The performance of the OpenOrca Platypus2 model, which was trained on data that included Python code, was not remarkable either.

Over half of the models identified at least one sample in the subset that did not contain a CWE (i.e., the model answered "no" to our query). Given that the Falcon 7B Instruct model found no CWEs in the highest number of files from the full dataset, it is not surprising that it identified 48 (96%) samples with no CWEs in this experiment.

Analysis of detected CWEs. Table 4 shows the number of unique CWEs and CWE combinations identified by the models during the code assessment of the samples. The LLaMA 2 7B and 13B models suggested the highest number of unique CWEs. Across all of the models, CWE-311, CWE-77, CWE-78, CWE-434 and CWE-319 were the top five most identified CWEs. CWE-311 (Missing Encryption of Sensitive Data), which occurs when data is improperly encrypted, was identified 21 times. CWE-77 and CWE-78 were detected 19 and 18 times, respectively. Both of these CWEs are concerned with variations of command injection, which can give an attacker a privilege or capability that they would not otherwise have.

When comparing the LLM assessments to the SecurityEval dataset, CWE-311, CWE-77, CWE-78, CWE-434, and CWE-319 were the most commonly identified CWEs that did not match the labelling used by SecurityEval. CWE-311 and CWE-77 were mismatched 21 and 19 times respectively, corresponding to the number of times they were identified across all of the models. However, CWE-78 was mismatched only 15 times, meaning that CWE-78 was correctly matched to the SecurityEval dataset three times since it was detected 18 times. Some of the LLMs that we evaluated learned from text to differentiate between secure and insecure code (e.g., code containing CWE-78), making it possible for LLMs to detect this form of weakness.

When comparing the results of the model assessments to the CodeQL scan, CWE-77, CWE-311, CWE-78, CWE-434 and CWE-122 were the most commonly identified CWEs that also did not match the CodeQL result for that corresponding file. CWE-77 was the most commonly mismatched CWE; mismatches occurred 13 times.

In Table 5 we present the agreement between each variety of model and our two baselines. Specifically, we show the percentage of evaluations done by each model type that detected the same CWE in a code sample as either one of our baselines. These percentage rates are counted separately for each baseline. For both comparisons, the baseline LLaMA 2 model performed the best in this metric with a 7.4% overlap with the SecurityEval baseline and a 10% agreement with the CodeQL evaluation. Interestingly, many of the models that were derivatives of LLaMA 2 performed much worse than the base model. We believe that this is due to the fine-tuning toward chat data of these derivative models, which reduces the weight of code analysis data in the training sets. To support this, the other two model types showing any overlap at all with the baselines were the Nous Hermes and the LLaMA 2 Guanaco QloRA models, both of which contain non-chat-based fine-tuning. In addition, with the exception of OpenOrca Platypus, all chat tuned models showed zero overlap. The original Guanaco model was also not chat trained, but it was based on the original LLaMA instead of LLaMA 2, which could account for its variance.

Table 6 shows the distribution of the models' assessments sorted by the 10 pillars of the Research Concepts view (CWE-1000) defined by the CWE community. In the CWE community, views are hierarchical structures used to organize weaknesses, and this view is intended to include every CWE weakness [11]. Pil-

lars exist within this view with the purpose of grouping weaknesses. The models identified numerous CWEs in the code samples, and we sort these CWEs into their top-level pillar entry to present a more cohesive relationship between the models and the CWEs they identified. We observe that LLaMA 2 7B identified 45 CWE-707s, which is 37% of the code samples it assessed. CWE-707 (Improper Neutralization) typically occurs when input (and in some cases, output) is not correctly validated to ensure that it is safe and will behave as expected when used in the code [10]. Additionally, LLaMA 2 13B identified 40 instances of CWE-664 (Improper Control of a Resource Through its Lifetime) and 30 instances of CWE-693 (Protection Mechanism Failure) during its code assessments. Together these two CWEs account for 58% of this model's assessments. Code contains the CWE-664 when it fails to properly create, use and destroy resources according to the resource's instructions [8]. CWE-693 occurs when a protection mechanism is missing, insufficient, or not applied in all relevant situations in the code [9].

CWE-664, CWE-693 and CWE-707 are similar in the sense that they can be represented in text as patterns. LLaMA 2 13B identified a high number of CWE-664 and CWE-693 rather than identifying small numbers of many different CWEs. We also see this trend with the LLaMA 2 7B model where it identified a high number of one CWE (CWE-707). Presumably, since the 13B model is exposed to more patterns than the 7B due to the size of its training data, it detects a wider range of CWEs at a high number compared to the rest of the dataset.

In other words, larger models should identify more variety of CWEs with less specificity (i.e., numbers should be more spread out and not as high). We see this when we examine the CWE breakdown for the LLaMA 2 7B and 13B models. The 7B model peaks higher since it identifies 45 files with the CWE-707 weakness, and the 13B model peaks twice with slightly lower numbers.

Interestingly, the LLaMA 2 13B model flagged three files that matched the SecurityEval labelling, but, due to the limited number of CWE rules defined by CodeQL, these files were not flagged by CodeQL. Two of these files contained CWE-259 (Use of Hard-coded Password) and one file contained CWE-200 (Exposure of Sensitive Information to an Unauthorized Actor). Similarly, the StarCoderPlus model correctly identified (according to the SecurityEval labelling) one file that contained CWE-321 (Use of Hard-coded Cryptographic Key). These three CWEs can be represented as text, and they are not CWEs that are currently detected by CodeQL according to the defined rules in CodeQL's repository on Github. These results suggest that LLMs and CodeQL can pair well together to detect a wider range of weaknesses in code.

5 Discussion

The use of CWEs is an attempt to formalize the effect of weakness in code, and it is a valuable way to standardize weakness across the community. As indicated by the differences in CWE identification between the models' assessments, the CodeQL scan, and the labelling of the SecurityEval dataset, identifying code

Table 3: Models' code assessments compared to the CodeQL scan

Model	**Provided CWE matched**	**CWE did not match**	**No response**	**No CWE found**	**Other response**
LLaMA 2 13B	4	44	0	2	0
StarCoderPlus	2	6	38	4	0
LLaMA 2 7B	1	32	3	8	6
LLaMA 2 7B Guanaco QLoRA	1	13	19	13	4
Nous Hermes LLaMA 2 13B	1	3	38	8	0
StarCoder	1	15	29	4	1
Vicuna 33B	0	0	50	0	0
Airoboros 13B GPT4-1.4	0	0	50	0	0
OpenOrca Platypus2 13B	0	11	2	37	0
LLaMA 2 13B Guanaco QLoRA	0	17	24	8	1
StableBeluga 7B	0	5	32	13	0
CodeGen2.5 7B Mono	0	0	50	0	0
Airoboros 7B GPT4-1.2	0	0	50	0	0
Guanaco 7B	0	0	50	0	0
Falcon 7B Instruct	0	0	2	48	0
Wizard Vicuna 7B Uncensored	0	0	50	0	0
Samantha 33B	0	0	50	0	0
Luna AI LLaMA Uncensored	0	3	15	32	0
StableBeluga 13B	0	4	15	31	0
Chronos Hermes 13B	0	0	50	0	0
LLaMA 2 Coder 7B	0	0	50	0	0
Samantha Falcon 7B	-	-	-	-	-
Wizard Vicuna 30B Uncensored	-	-	-	-	-
LLaMA2 70B Guanaco QLoRA	-	-	-	-	-
StableBeluga2 70B	-	-	-	-	-

weaknesses can be challenging. Programming languages are continually evolving; subsequently, code weaknesses also evolve. Using LLMs to detect weaknesses in code allows us to shift away from having to continually redefine version- and code-dependent weaknesses.

Specifically, our findings emphasize several important points:

- **Model size and parameter number.** Our analysis shows that the size of a model and its number of parameters is not a reflection of a model's ability to assess weakness in code. For example, the LLaMA 2 13B model outperformed other larger models.
- **Best performing model.** Out of the evaluated models, LLaMA 2 13B performed the best. Overall, it was robust in response to the provided code samples, and it provided a CWE for 105 code samples. Its assessments were most in line with the SecurityEval dataset's CWE labelling and the CodeQL results. It also provided the highest number of unique CWEs.
- **Model training datasets.** Models that were trained on more generalized data tended to identify a higher number of CWEs that were in line with

Table 4: The number of unique CWEs identified by models during code assessment

Model	Unique CWEs
LLaMA 2 13B	38
LLaMA 2 7B	35
StarCoder	24
LLaMA 2 13B Guanaco QLoRA	19
LLaMA 2 7B Guanaco QLoRA	17
OpenOrca Platypus2 13B	17
StarCoderPlus	12
Luna AI LLaMA Uncensored	10
Nous Hermes LLaMA 2 13B	9
StableBeluga 13B	5
StableBeluga 7B	4
CodeGen2.5 7B Mono	1
Airoboros 13B GPT4-1.4	0
Vicuna 33B	0
Airoboros 7B GPT4-1.2	0
Guanaco 7B	0
Falcon 7B Instruct	0
Wizard Vicuna 7B Uncensored	0
Samantha 33B	0
Chronos Hermes 13B	0
LLaMA 2 Coder 7B	0
Samantha Falcon 7B	-
Wizard Vicuna 30B Uncensored	-
LLaMA2 70B Guanaco QLoRA	-
StableBeluga2 70B	-

the SecurityEval dataset categorization and the CodeQL analysis. Models that were specifically trained on code displayed mediocre performances with the exception of the StarCoderPlus model. However, this model was still outperformed by the LLaMA 2 13B model. Hence, it appears that models equipped to solve coding-related questions do not have sufficient reasoning to identify code weaknesses.

- **LLM code evaluation.** Detecting weakness is a form of code evaluation and not a property of code. Developing models that can identify code weaknesses requires a training dataset of both code data as well as non-coding data related to code evaluation (i.e., a mixed model).
- **CWE representation.** Models often detected the same types of CWEs, which were mainly patterns of CWE concepts represented as text. CWEs that are not represented as text, and therefore must be inferred by the LLM, are not easily detected by LLMs. This finding suggests that LLMs require more knowledge about code evaluation to reason about code weakness.
- **Pairing LLMs with static analysis tools.** As we saw in our analysis, two LLMs detected CWEs that were outside the scope of CodeQL. Tool

Table 5: Detection rate for each model type

Model Type	Parent Model	Rate (SE)	Rate (CodeQL)
LLaMA 2	NA	7.40%	10%
Luna AI LLaMA	LLaMA 2	0.00%	0%
Nous Hermes	LLaMA 2	0.80%	2%
Vicuna	LLaMA	0.00%	0%
Airoboros*	LLaMA	0.00%	0%
OpenOrca Platypus*	LLaMA 2	1.70%	0%
LLaMA 2 Guanaco QloRA	LLaMA 2	0.80%	2%
StableBeluga	LLaMA 2	0.00%	0%
Guanaco	LLaMA	0.00%	0%
Falcon	NA	0.00%	0%
Wizard Vicuna	LLaMA + WizardLM	0.00%	0%
Samantha	GPT-4	0.00%	0%
Chronos Hermes*	LLaMA + LLaMA 2	0.00%	0%
Starcoder*	StarCoderBase	0.80%	2%
StarcoderPlus*	StarCoder	2.50%	4%
LLaMA 2 Coder	LLaMA 2	0.00%	0%
CodeGen2.5*	CodeGen2	0.00%	0%

* Presence of code samples, data from technical forums, or coding documentations in the training or fine-tuning stages

pairings such as this can potentially detect a wider range of code weaknesses before significant problems arise. Since the process of defining the vulnerable patterns used by static analysis tools is time consuming and prone to errors, using LLMs to complement the existing tools may expedite the process of identifying weaknesses in code.

5.1 Limitations

Although CodeQL offers vulnerability detection at an industry standard, it does have some limitations. CodeQL is composed of queries defined only for a small range of CWEs for Python code, and it does not support detection for other CWEs that can affect Python code. As a result, this limits our ability to fully compare the models' assessments to the full range of CWEs.

Hugging Face provides several popular models for use within the AI community. Despite this accessibility, some models did not successfully load on our system despite our system meeting the model's requirements. It is unclear if there was an internal issue with the model, or another reason entirely. Four of our selected models (three of which had parameters greater than or equal to 30B) did not load and were not evaluated, therefore limiting our ability to evaluate larger models.

6 Conclusion

In this study, we compared the weakness assessment capabilities of various LLMs. Our analysis sheds light on how different types of models detect code weakness.

Table 6: Breakdown of model responses by CWE pillars

Model	CWE 284	CWE 435	CWE 664	CWE 682	CWE 691	CWE 693	CWE 697	CWE 703	CWE 707	CWE 710	No corresponding CWE pillar	No CWE	Total
CodeQL	1	0	6	0	0	8	0	0	3	8	24	71	121
LLaMA 2 13B	4	0	40	1	1	30	0	0	14	1	14	16	121
LLaMA 2 7B	2	3	17	0	4	1	5	1	45	0	4	39	121
LLaMA 2 7B Guanaco QLoRA	2	0	5	2	0	1	0	0	4	4	6	97	121
LLaMA 2 13B Guanaco QLoRA	2	0	13	0	0	8	1	0	0	1	4	92	121
Luna AI LLaMA Uncensored	1	0	3	0	0	1	0	1	2	0	2	111	121
Nous Hermes LLaMA 2 13B	1	0	3	0	0	6	1	0	1	0	0	109	121
OpenOrca Platypus2 13B	1	0	8	0	2	5	0	0	3	0	7	95	121
Vicuna 33B	0	0	0	0	0	0	0	0	0	0	0	121	121
Airoboros 13B GPT4-1.4	0	0	0	0	0	0	0	0	0	0	0	121	121
CodeGen2.5 7B Mono	0	0	0	0	0	0	0	0	1	0	0	120	121
StableBeluga 7B	0	0	5	0	0	4	0	1	0	0	0	111	121
Airoboros 7B GPT4-1.2	0	0	0	0	0	0	0	0	0	0	0	121	121
Guanaco 7B	0	0	0	0	0	0	0	0	0	0	0	121	121
Falcon 7B Instruct	0	0	0	0	0	0	0	0	0	0	0	121	121
StarCoderPlus	0	0	11	0	0	0	0	0	1	0	8	101	121
Wizard Vicuna 7B Uncensored	0	0	0	0	0	0	0	0	0	0	0	121	121
LLaMA 2 Coder 7B	0	0	0	0	0	0	0	0	0	0	0	121	121
Samantha 33B	0	0	0	0	0	0	0	0	0	0	0	121	121
StarCoder	0	0	10	0	0	1	0	0	0	1	31	78	121
StableBeluga 13B	0	0	2	0	0	2	0	0	1	0	0	116	121
Chronos Hermes 13B	0	0	0	0	0	0	0	0	0	0	0	121	121
Samantha Falcon 7B	-	-	-	-	-	-	-	-	-	-	-	-	-
Wizard Vicuna 30B Uncensored	-	-	-	-	-	-	-	-	-	-	-	-	-
LLaMA 2 70B Guanaco QLoRA	-	-	-	-	-	-	-	-	-	-	-	-	-
StableBeluga2 70B	-	-	-	-	-	-	-	-	-	-	-	-	-

We asked each model to assess a dataset of Python code and detect any CWEs. The models' assessments were compared to the original CWE labelling assigned to the code samples by the SecurityEval authors. We further explored the models' abilities to identify weaknesses by comparing their assessments to the results of CodeQL, a widely used static analysis tool.

We show that a model's success in detecting CWEs is more influenced by the content of its training dataset rather than its size and parameter number alone. Overall, the LLaMA 2 models, which were trained on general datasets, outperformed models that were specifically trained on code samples. Additionally, some of the models in our dataset did not have sufficient knowledge of code and CWEs, resulting in a high number of false negatives. This result suggests that models require a knowledge of code as well as an understanding of how code is evaluated to reliably detect code weaknesses.

Out of the models in our evaluation, only a small portion of models identified a CWE in at least one code sample. Upon closer examination, certain CWEs were frequently detected; many of which were CWEs that are represented as text and can therefore be detected by the model. CWEs that are not represented as text (i.e, CWEs that are detected through model inference) are not easily detectable.

Our analysis of popular LLMs demonstrates that certain types of models are better suited for use in code weakness detection. Ultimately, the goal with detecting weakness in code is to detect as many weaknesses as possible before such weaknesses cause significant problems for end users. Pairing LLMs with static analysis tools such as CodeQL to detect weaknesses in code is the next step toward comprehensive code analysis.

References

1. Adamson, V., Bägerfeldt, J.: Assessing the effectiveness of chatgpt in generating python code (2023)
2. Ahmed, I., Kajol, M., Hasan, U., Datta, P.P., Roy, A., Reza, M.R.: Chatgpt vs. bard: A comparative study. UMBC Student Collection (2023)
3. Airoboros: Airoboros: using large language models to fine-tune large language models, https://github.com/jondurbin/airoboros
4. Austism: Chronos-hermes-13b, https://huggingface.co/Austism/chronos-hermes-13b
5. Bilgin, Z., Ersoy, M.A., Soykan, E.U., Tomur, E., Çomak, P., Karaçay, L.: Vulnerability prediction from source code using machine learning. IEEE Access **8**, 150672–150684 (2020)
6. Bull, C., Kharrufa, A.: Generative ai assistants in software development education: A vision for integrating generative ai into educational practice, not instinctively defending against it. IEEE Software (2023)
7. Corporation, M.: Common weakness enumeration, https://cwe.mitre.org/
8. CWE: Cwe-664: Improper control of a resource through its lifetime, https://cwe.mitre.org/data/definitions/664.html
9. CWE: Cwe-693: Protection mechanism failure, https://cwe.mitre.org/data/definitions/693.html
10. CWE: Cwe-707: Improper neutralization, https://cwe.mitre.org/data/definitions/707.html
11. CWE: Cwe view: Research concepts, https://cwe.mitre.org/data/definitions/1000.html
12. E. Nijkamp, H. Hayashi, Y.Z.C.X.: Codegen2.5: Small, but mighty, https://blog.salesforceairesearch.com/codegen25/
13. Face, H.: The ai community building the future, https://huggingface.co/
14. Fu, Yao; Peng, H., Khot, T.: How does gpt obtain its ability? tracing emergent abilities of language models to their sources. Yao Fu's Notion (Dec 2022), https://yaofu.notion.site/How-does-GPT-Obtain-its-Ability-Tracing-Emergent-Abilities-of-Language-Models-to-their-Sources-b9a57ac0fcf74f30a1ab9e3e36fa1dc1
15. Hartford, E.: Samantha-33b, https://huggingface.co/ehartford/samantha-33b
16. Hartford, E.: Wizard-vicuna-7b-uncensored, https://huggingface.co/ehartford/Wizard-Vicuna-7B-Uncensored
17. Index, P.P.: llama2-wrapper 0.1.12, https://pypi.org/project/llama2-wrapper/
18. Institute, T.I.: Falcon 7b instruct, https://huggingface.co/tiiuae/falcon-7b-instruct
19. Ji, B.: Vicunaner: Zero/few-shot named entity recognition using vicuna. arXiv preprint arXiv:2305.03253 (2023)
20. Kande, R., Pearce, H., Tan, B., Dolan-Gavitt, B., Thakur, S., Karri, R., Rajendran, J.: Llm-assisted generation of hardware assertions (2023)
21. Khoury, R., Avila, A.R., Brunelle, J., Camara, B.M.: How secure is code generated by chatgpt? arXiv preprint arXiv:2304.09655 (2023)
22. Lab, G.S.: Codeql, https://codeql.github.com/
23. Lee, A.N., Hunter, C.J., Ruiz, N.: Platypus: Quick, cheap, and powerful refinement of llms (2023)
24. Li, R., Allal, L.B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T.Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M.H., Umapathi, L.K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., Murthy, R., Stillerman, J., Patel, S.S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S.,

Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C.J., Dolan-Gavitt, B., Contractor, D., Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C.M., Hughes, S., Wolf, T., Guha, A., von Werra, L., de Vries, H.: Starcoder: may the source be with you! (2023)
25. Mahan, D., Carlow, R., Castricato, L., Cooper, N., Laforte, C.: Stable beluga models, https://huggingface.co/stabilityai/StableBeluga2
26. Meta: Meta and microsoft introduce the next generation of llama, https://about.fb.com/news/2023/07/llama-2/
27. Nayak, A., Timmapathini, H.P.: Llm2kb: Constructing knowledge bases using instruction tuned context aware large language models. arXiv preprint arXiv:2308.13207 (2023)
28. Open-Orca: Openorca-platypus2-13b, https://huggingface.co/Open-Orca/OpenOrca-Platypus2-13B
29. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R.: Asleep at the keyboard? assessing the security of github copilot's code contributions. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 754–768 (2022)
30. Pearce, H., Tan, B., Ahmad, B., Karri, R., Dolan-Gavitt, B.: Examining zero-shot vulnerability repair with large language models. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 2339–2356. IEEE (2023)
31. Research, N.: Nous-hermes-llama2-13b, https://huggingface.co/NousResearch/Nous-Hermes-Llama2-13b
32. Romero, M.: llama-2-coder-7b (revision d30d193) (2023), https://huggingface.co/mrm8488/llama-2-coder-7b
33. Sandoval, G., Pearce, H., Nys, T., Karri, R., Garg, S., Dolan-Gavitt, B.: Lost at c: A user study on the security implications of large language model code assistants. In: USENIX (2023)
34. Sharma, S., Sodhi, B.: Calculating originality of llm assisted source code (2023)
35. Siddiq, M.L., Santos, J.C.S.: Securityeval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques. In: Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4PS22) (2022). https://doi.org/10.1145/3549035.3561184
36. SonarSource: Sonarsource static code analysis, https://rules.sonarsource.com/
37. Surameery, N.M.S., Shakor, M.Y.: Use chat gpt to solve programming bugs. International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290 **3**(01), 17–22 (2023)
38. Taecharungroj, V.: "what can chatgpt do?" analyzing early reactions to the innovative ai chatbot on twitter. Big Data and Cognitive Computing **7**(1), 35 (2023)
39. Tap-M: Luna ai llama uncensored, https://huggingface.co/Tap-M/Luna-AI-Llama2-Uncensored
40. Yamaguchi, F., Rieck, K., et al.: Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In: 5th USENIX Workshop on Offensive Technologies (WOOT 11) (2011)
41. Yetiştiren, B., Özsoy, I., Ayerdem, M., Tüzün, E.: Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt (2023)