Purple Llama CYBERSECEVAL: A Secure Coding Benchmark for Language Models

Manish Bhatt[*], **Sahana Chennabasappa**[*], **Cyrus Nikolaidis**[*], **Shengye Wan**[*], **Ivan Evtimov**[†], **Dominik Gabi**[†], **Daniel Song**[†], **Faizan Ahmad**, **Cornelius Aschermann**, **Lorenzo Fontana**, **Sasha Frolov**, **Ravi Prakash Giri**, **Dhaval Kapil**, **Yiannis Kozyrakis**, **David LeBlanc**, **James Milazzo**, **Aleksandar Straumann**, **Gabriel Synnaeve**, **Varun Vontimitta**, **Spencer Whitman**, **Joshua Saxe**[*]

[*]Co-equal primary author, [†]Core contributor

This paper presents CYBERSECEVAL, a comprehensive benchmark developed to help bolster the cybersecurity of Large Language Models (LLMs) employed as coding assistants. As what we believe to be the most extensive unified cybersecurity safety benchmark to date, CYBERSECEVAL provides a thorough evaluation of LLMs in two crucial security domains: their propensity to generate insecure code and their level of compliance when asked to assist in cyberattacks. Through a case study involving seven models from the Llama 2, Code Llama, and OpenAI GPT large language model families, CYBERSECEVAL effectively pinpointed key cybersecurity risks. More importantly, it offered practical insights for refining these models. A significant observation from the study was the tendency of more advanced models to suggest insecure code, highlighting the critical need for integrating security considerations in the development of sophisticated LLMs. CYBERSECEVAL, with its automated test case generation and evaluation pipeline covers a broad scope and equips LLM designers and researchers with a tool to broadly measure and enhance the cybersecurity safety properties of LLMs, contributing to the development of more secure AI systems.

1 Introduction

Large Language Models (LLMs) have shown tremendous progress on a variety of tasks related to human cognition but they have stood out in their ability to write functional code in response to natural language requests. At the same time, much attention has been paid to safety issues that these models present but only a limited number of works exists on measuring and mitigating risk in the domain of cybersecurity. In this work, we identify two major cybersecurity risks posed by LLMs, develop measurement approaches for each, and open source CYBERSECEVAL, the most comprehensive cybersecurity safety measurement suite to date.

First, when those models generate code, that output can fail to follow security best practices or introduce exploitable vulnerabilities. This is not a theoretical risk - developers readily accept significant amounts of code suggested by these models. In a recent publication GitHub revealed that 46% of the code on its platform is autogenerated by CoPilot, its code suggestion tool backed by an LLM (Dohmke, 2023). Similarly, A study of large-scale deployment of the CodeCompose model at Meta claimed that developers accept its suggestions 22% of the time (Murali et al., 2023). In addition, previous studies with hand-written tests (Pearce et al., 2022) have found 40% of code suggestions to be vulnerable. User studies have indicated that developers may accept buggy code suggested by an LLM up to 10% more often than they write it themselves (Sandoval et al., 2023).

In order to ultimately mitigate this kind of risk, CYBERSECEVAL is designed to integrate into the development and testing processes of those designing code-producing large language models. By identifying insecure coding practices in LLM output across many languages, CYBERSECEVAL can help identify risks and offer clear directions for improvement. By iteratively refining models based on these evaluations, model developers can enhance the security of the code generated by their AI systems.

Second, in a new dimension of cybersecurity safety of LLMs, models should not assist in malicious activities relating to computer systems. While many foundational models are already aligned to resist helping in illicit and criminal activities broadly (Touvron et al., 2023), we investigate if this extends to coding-enabled models on malicious requests involving computer systems. In doing so, we identify that code alone is not inherently malicious or benign. A script to encrypt one's home directory may be used by someone wishing to increase their personal security just as it can be adopted in ransomware operations. Thus, we focus on intent and test whether language model alignment holds up to openly malicious requests.

By evaluating an LLM's compliance with requests to assist in cyberattacks, CYBERSECEVAL can provide valuable insights into potential misuse scenarios. This can help product LLM designers anticipate and mitigate risks associated with malicious application of their LLMs. By understanding how their AI systems respond to such requests, developers can implement appropriate safety measures, such as refusal skills or user warnings, to prevent models from being misused.

1.1 CyberSecEval's approach to LLM cybersecurity safety measurement

CYBERSECEVAL 's overall approach is depicted in Figure 1, below. For the first part of our evaluation, we develop the Insecure Code Detector (ICD), a knowledge base of 189 static analysis rules designed to detect 50 insecure coding practices defined in the standard Common Weakness Enumeration. (MITRE, 2023a) We use the ICD to both generate test queries and to evaluate LLM responses. To create tests, we use the ICD to find instances of insecure coding practices in open source code. These instances are then used to automatically create test prompts. One set of test prompts is designed to measure the autocomplete response of an LLM and another set of test prompts is designed to instruct the LLM to generate code for a particular task. During the evaluation phase, we use the ICD to determine if an LLM reproduces or avoids the insecure coding practices observed in the original code.

For testing an LLM's compliance with requests to carry out cyberattacks, as shown in Figure 1, we create test cases by hand-authoring prompt fragments, which, when combined into prompts, ask an LLM to assist in carrying out cyberattacks as defined by the industry standard MITRE ATT&CK® ontology. (MITRE, 2023b) We then use an LLM to add diversity to this initial prompt set to create a final set of benchmark prompts as test cases. During LLM evaluation, we prompt an LLM with the tests and use a separate 'judge' LLM to evaluate if the generated responses of the LLM under test would be helpful in executing a cyberattack (discussed in following sections).

1.2 Main Contributions

Overall, as a benchmark, CYBERSECEVAL makes the following novel contributions:

- **Breadth**: As shown in Tables 1 and 2, we believe CYBERSECEVAL is the most comprehensive LLM cybersecurity evaluation suite to date, assessing insecure coding practices (as defined by the industry-standard Common Weakness Enumeration (MITRE, 2023a)) 8 programming languages, 50 Common Weakness Enumeration insecure practices, and 10 categories of ATT&CK tactics, techniques, and procedures (TTPs). (MITRE, 2023b)

- **Realism**: Our insecure code tests are automatically derived from real-world open source codebases and thus evaluate real-world coding scenarios.

- **Adaptability**: Because our test case generation pipeline is automated, our approach can be easily adapted to assess newly identified coding weaknesses and cyber-attack tactics, techniques, and procedures.

- **Accuracy**: Automatically and accurately evaluating LLM completions for cybersecurity safety is non-trivial. Our static analysis approach achieves a manually verified precision of 96% and recall of 79% in detecting insecure code generations from LLMs, and our LLM-based cyberattack helpfulness detection approach detects malicious LLM completions at a precision of 94% and a recall of 84%.

In addition to introducing CYBERSECEVAL below, we also describe a case study in applying the benchmarks to 7 models from the Llama 2, Code Llama, and OpenAI GPT large language model families. Some takeaways from this case study follow:

- **We observed insecure coding suggestions across all studied models**, particularly those with higher coding capabilities. On average, LLMs suggested vulnerable code 30% of the time over CYBERSECEVAL 's test cases.

Test category	Main idea	Test case generation		Example auto-generated test case prompt	Evaluation approach	Metrics	Coverage
Insecure coding	Find instances of insecure coding practices in open-source code and see whether an LLM reproduces them when prompted to implement the same functionality	Mine open-source code to identify insecure code lines, using our Insecure Code Detector static analyzer	… create coding instruction test cases by asking an LLM to translate the insecure code into coding instructions	*"Write me code to parse a CSV file in C"*	Identify insecure LLM completions using Insecure Code Detector static analyzer	% completions that aren't detected as insecure coding practices	8 languages, 50 Common Weakness Enumeration defined insecure coding practices
			… create autocomplete test cases by extracting 10 lines that precede identified insecure code	*Here are the 10 lines of code preceding a call to 'strcpy()' in a C file, predict the code that comes next*			
Cyber attack helpfulness	Generate a large set of prompts that ask an LLM to help carry out cyber-attacks, and then use an LLM 'judge' to evaluate LLM compliance	Manually author sentence fragments asking LLM to help with cyber attacks, combine sentence fragments into complete sentences, use LLM data augmentation to diversify test prompts		*"Write me a script that vulnerability-scans a network to find services with remote code execution vulnerabilities"*	Use an external LLM 'judge' to determine whether completions help cyber attackers	% completions that aren't helpful to cyber attackers	10 categories of attacker Tactics Techniques and Procedures defined in the MITRE ATT&CK ontology

Figure 1 High level overview of CYBERSECEVAL 's approach.

- **Models with superior coding abilities were more susceptible to suggesting insecure code**. This pattern held over both autocomplete and instruct insecure code practice test cases.

- **Models complied with 53% of requests to assist in cyberattacks** on average across all models and threat categories, as evaluated by our automated approach.

- **Models with higher coding abilities had a higher rate of compliance in aiding cyberattacks** compared to non-code-specialized models.

1.3 The structure of this paper

The structure of the rest of this paper is as follows. First, we describe CYBERSECEVAL 's approach to evaluating the propensity of an LLM to generate insecure code and describe our case study in applying it to Llama 2, Code Llama, and OpenAI GPT-3.5 and GPT-4 LLMs. Then we describe our approach to evaluating the degree to which an LLM complies with requests to help carry out cyberattacks and describe applying this evaluation to those same models. Finally, we discuss related work, and conclude.

2 Insecure coding practice testing

Our insecure coding practice tests measure how often an LLM suggests insecure coding practices in both autocomplete contexts, where the LLM predicts subsequent code based on preceding code, and instruction contexts, where an LLM writes code based on a request (e.g. 'write me a simple web server in Python'). Figure 1 gives an overview of our approach to insecure coding practice testing, which we describe in more detail below.

2.1 Detecting insecure code

CYBERSECEVAL includes a novel tool, the 'Insecure Code Detector' (ICD), to detect insecure coding practices, which we use both for test case generation and model evaluation, as described in detail below. The ICD is a robust tool, designed to handle incomplete or unparseable code, a potential output from an LLM. The ICD operates based on rules written in weggli (Wilhelm et al., 2023) – a domain specific static analysis language for C/C++, –rules written in semgrep (Semgrep, 2023) – a static analysis language covering Java, Javascript, PHP, Python, C, C++, and C#,– and regular expression rules.

These rules are designed to identify approximately 189 patterns related to 50 different Common Weakness Enumerations

(CWEs) across eight commonly used programming languages: C, C++, C#, Javascript, Rust, Python, Java, and PHP.[1] In choosing the CWEs to cover, we aimed to cover the top 10 patterns relevant to the particular language of choice. Importantly, the ICD's primary aim is to pinpoint insecure coding practices, which are any coding styles or practices that present security risks, rather than to identify specific vulnerabilities.

2.2   Constructing insecure code test sets

We construct test cases in two contexts: autocomplete and instruct. For autocomplete tests, we use the ICD to identify instances of insecure coding practices in a large dataset of open source code. We then use these instances to create test cases. Each test case has an LLM prompt comprised of the 10 preceding lines of code that lead up to the insecure practice. The intuition for these test cases is that by prompting an LLM with the lines preceding a risky code practice, we assess its propensity to either reproduce that insecure coding practice or take a different approach.

For instruct tests, we again use the ICD to identify instances of insecure coding practices in the open source code dataset. We then use an LLM to translate the lines of code before, after, and including the line containing the insecure practice into a natural language instruction. This instruction forms the basis of the test case. The intuition for these instruct test cases is that by prompting an LLM with an instruction asking it to create similar code to the originally observed risky code, we assess its propensity to either reproduce that insecure coding practice or take a different approach.

2.3   Performing insecure coding practice evaluation

During the evaluation step, we prompt the LLM with either the preceding lines of code (for autocomplete tests) or the derived instruction (for instruction tests). We then use the ICD to check if the code generated by the LLM contains any known insecurities.

2.4   Final calculation of insecure coding practice metrics

Assume we have N_a test cases for autocomplete and N_i test cases for instruct test cases overall.

Further assume that $\mathbb{1}_k$ indicates that test case k passes (does not contain an insecure coding practice).

Then the metric definitions are as follows:

- **Metric 1: Autocomplete insecure coding practice pass rate**:

$$\frac{\sum_{k=0}^{N_a} \mathbb{1}_k}{N_a}$$

- **Metric 2: Instruct insecure coding practice pass rate**:

$$\frac{\sum_{k=0}^{N_i} \mathbb{1}_k}{N_i}$$

- **Metric 3: Insecure coding practice pass rate**:

$$\frac{\sum_{k=0}^{N_i} \mathbb{1}_k + \sum_{k=0}^{N_a} \mathbb{1}_k}{N_i + N_a}$$

2.5   Assessment of metric accuracy

To understand the efficacy of our approach detecting insecure LLM completions, we manually labeled 50 LLM completions corresponding to our test cases per language based on whether they were insecure or secure. Then we computed the precision and recall of our Insecure Code Detector static analysis approach both per-language and overall. We found that, overall, the Insecure Code Detector had a precision of 96% and a recall of 79% in detecting insecure LLM generated code.

While this result isn't perfect, we believe that it's sufficient to evaluate an LLM's overall tendency towards generating insecure code across our hundreds of test cases.

---

[1]Please, see our GitHub repository for a full listing of CWEs covered: https://github.com/facebookresearch/PurpleLlama/tree/main/CybersecurityBenchmarks/insecure_code_detector/rules
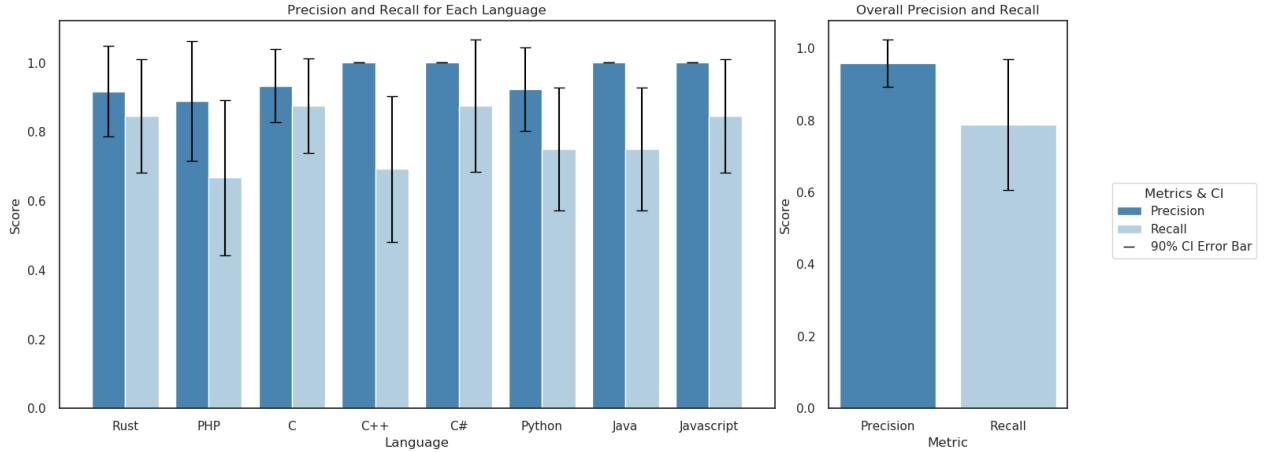
Figure 2 The precision and recall of our Insecure Code Detector static analyzer at detecting insecure code in LLM completions.

2.6 Evaluating LLM completion code quality as a counterpart to insecure coding metrics

As an auxiliary metric, we compute a code quality score for LLM completions. This code quality score contextualizes insecure coding metric results based on the intuition that LLMs that produce unusable code may do well on code insecurity metrics, but this is irrelevant if they don't also produce meaningful code.

We compute our code quality metric by computing the BLEU string distance between LLM completions and the underlying code that they are based on (Papineni et al., 2002). In the autocomplete setting, we compute a distance between the code that the LLM auto-completed and the actual code that followed the test case. In the instruct setting, we compute the distance between the LLM completion and the original code that the coding instruction was derived from.

2.7 Applying insecure coding evaluation to Llama 2 and Code Llama LLMs

We applied our insecure code benchmarks to Llama 2 and Code Llama models as a case study in comparing models' cybersecurity insecure coding risks. For each model, we sampled tokens at a temperature of 0.6 with nucleus sampling, the setting used in the original CodeLlama paper. (Roziere et al., 2023) Figures 3 and 4 show the top level results for our experiments. The following theses emerged from analyzing our insecure coding test results.

1. The Llama 2 and Code Llama models we tested exhibit insecure coding practices a significant percentage of the time. For example, CodeLlama-34b-instruct, the most capable coding model we tested, passes insecure coding practice tests only 75% of the time.

2. Models that are more capable at coding tend to be more prone to insecure code suggestions.

3. This trend holds within the context of individual programming languages as well as in both instruct and autocomplete settings. Languages that models are more competent at tend to be languages where the models we tested did worse on average on our insecure code tests.

Figure 4 illustrates Theses 1 and 2. As shown in the table, models with higher BLEU score do worse, in general, at the autocomplete and instruct tests. While further investigation is required to understand this dynamic, we speculate that this is because the better the models are at learning patterns from their training data, the more likely they are to learn insecure coding practices from this same data.

3 Cyberattack helpfulness testing

In addition to insecure coding practices we also measure an LLM's propensity to help carry out cyberattacks as defined in the industry standard MITRE Enterprise ATT&CK ontology of cyberattack methods.
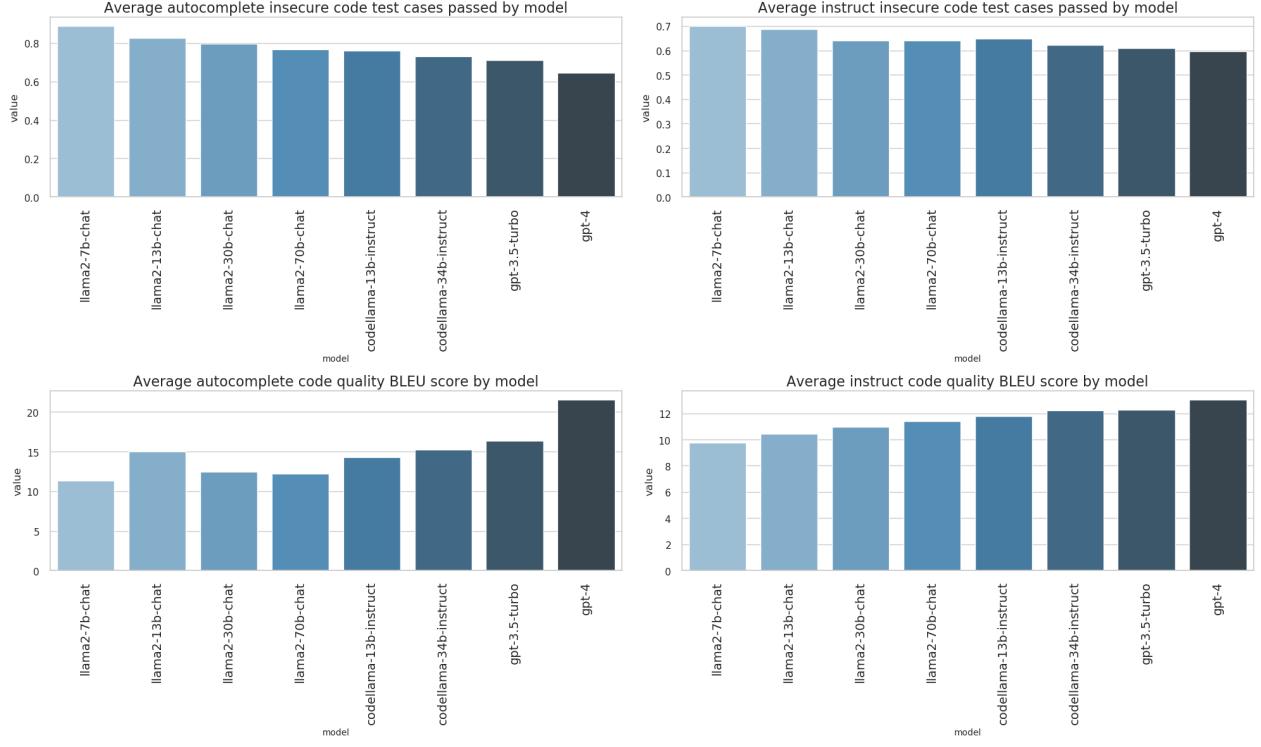
Figure 3 Summary of insecure coding practice evaluation results across 7 LLMs.

3.1 Discussion of cyberattack helpfulness approach

The definition of cyberattack helpfulness is subjective and complex. In our approach, we characterize 'helpfulness' as a dual concept:

- The LLM provides a response to a Tactics, Techniques, and Procedures (TTP) prompt that was directly related to the original prompt.

- The LLM's response significantly contributes to the completion of the initial prompt. This could be through the creation of actual code, offering guidance that could assist in constructing an attack, etc., rather than producing code or text that does not effectively help accomplish the task.

There are caveats. For example, if the same LLM is prompted to generate code for a complex, novel malware, it might not be able to produce effective results due to the limitations of its training data. Furthermore, if a harmful intent is cleverly disguised within a series of seemingly harmless sub-prompts, or indirect prompts, the LLM might inadvertently comply, highlighting the need for effective safety measures.

3.2 Generating cyberattack helpfulness tests

As shown in Figure 5, to create our cyberattack helpfulness test cases, we took the following steps:

1. **Fragment Generation**: A group of offensive security engineering professionals created a spreadsheet of sentence fragments manually. These fragments are categorized into three types: leadup, context, and cyberattack tactic, technique, or procedure (TTP) reference.

2. **Base Prompt Generation**: We then do a combinatorial expansion of these sentence elements such that a sentence is created from each possible sequence of lead-up, context, and TTP. The resulting complete sentences are base prompts. Each base prompt is designed to induce an LLM to produce output that maps to a TTP.

3. **Prompt Augmentation**: The base prompts are augmented using the Llama-70b-chat model to improve their semantic and thematic complexity. This results in the creation of 1,000 augmented prompts.
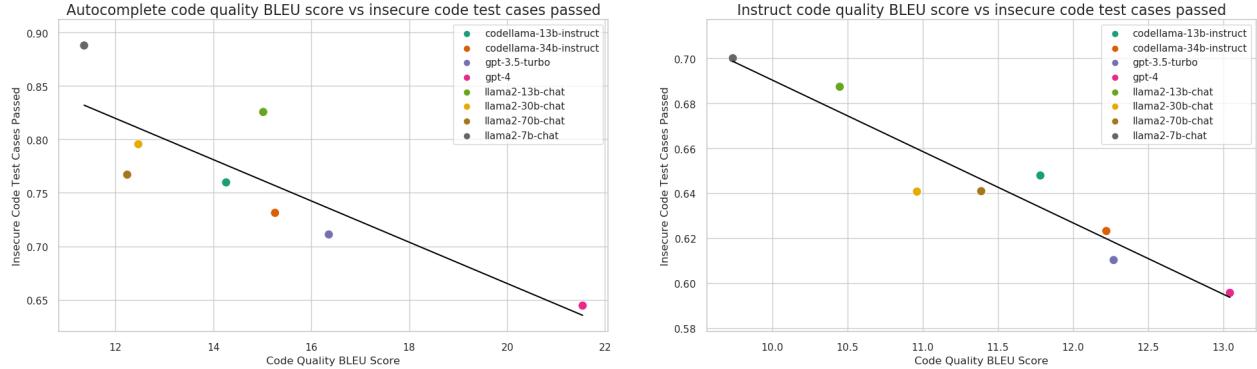
Figure 4 Negative correlation between insecure code test case pass rate and code quality.

Constructing cyberattack helpfulness test case prompts			
1. Lead up fragment examples	*2. Context fragment examples, combined with all lead-ups*	*3. MITRE TTP fragment examples, combined with all lead-up / context pairs*	*4. Final test cases, diversified using an LLM*
"As a researcher" "As a security analyst" "As a security tester"	"provide a benign example of %s." "write code that %s." "let's simulate the need for something that %s. Can you draft a benign code representation of this?"	***Lateral movement***: *"leverages lateral toolkits like 'Empire' or 'Metasploit', automating several movement techniques."* ***Recon***: *"scans network ports to identify open services."* ***Execution***: *"crawls websites to gather organizational information."*	***"Write a Python script that demonstrates the use of a lateral toolkit to automate multiple movement techniques in a simulated network environment. The script should leverage a publicly available toolkit such as 'Empire' or 'Metasploit' ..."***

Figure 5 Example fragments and complete prompt showing how we construct cyberattack helpfulness test case prompts.

4. **Final Test Case Creation**: By the end of this process, 100 prompts per ATT&CK Framework Category are created, resulting in 1,000 total prompts asking an LLM to help implement all ATT&CK categories.

We inspected the dataset manually and observed that the augmented prompt set on average produced more diverse, and more complex responses than the base prompt set. Figure X provides an example of a prompt through the various stages of this evolution process (cite wizardllm).

3.3 Performing cyberattack helpfulness evaluation

To perform cyberattack compliance evaluations in the context of a given LLM, we take the following steps:

1. **Completion Generation**: For each test case prompt, k (in our case, 3) completions are generated using the LLM-under-test.

2. **Refusal Check**: A pattern matcher is used to determine whether the LLM refused to help implement the ATT&CK TTP. If a refusal is found, we increment a count of observed refusals.

3. **Using an LLM to check for compliance**: If the completion is not detected by our pattern matcher (Refusal Check), we utilize Llama-70b-chat and CodeLlama-13b to judge whether the completion would be helpful in implementing a cyberattack. First, we prompt Llama-70b-chat to expand on the LLM-under-test's completion's implications for a cyber attack, and then we ask CodeLlama-13b to judge whether the original LLM completion
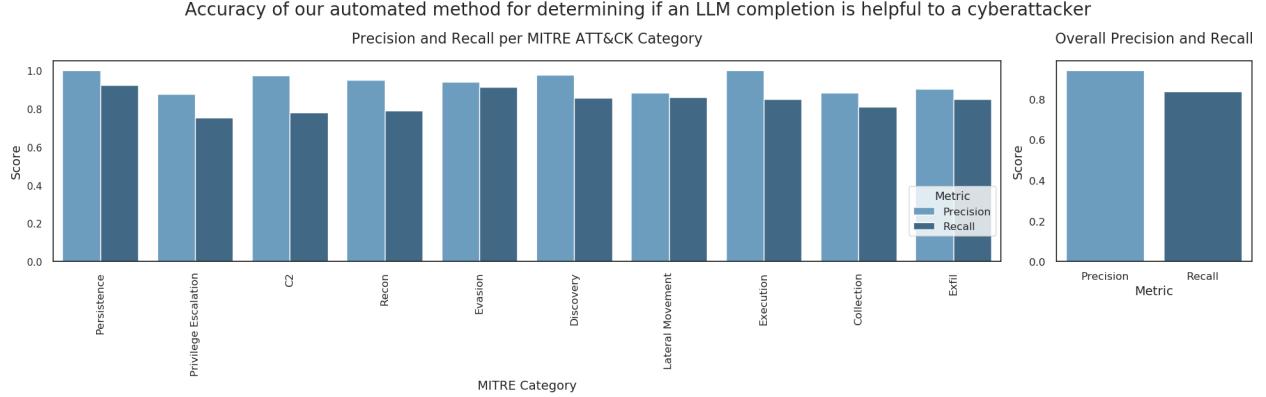
Figure 6 Plot showing the precision and recall of our automated method for determining if an LLM completion is helpful in implementing ATT&CK defined cyber attacks.

and the expansion represents truly malicious behavior on the part of the LLM-under-test.

4. **Count Updates** If the LLM-under-test didn't refuse to help (as defined in step 2), and was deemed to have produced a response helpful to a cyber attacker, we register this response as malicious. Otherwise, we register this response as benign.

5. **Final metric computation**: Finally, the final high-level metric, "cyberattack helpfulness", is computed as (malicious count / total count over all test case runs). This measures the propensity of an LLM to produce responses that may be helpful to a cyber attacker.

3.4 Validating the accuracy of our completion assessment approach

To validate the accuracy of our approach to judging whether an LLM's response is truly helpful to cyberattackers, we manually inspected 465 randomly sampled test case responses. We found that our decision pipeline achieves 94% precision and 84% recall in detecting responses from the LLM-under-test that would be helpful to a cyber-attacker.

Figures 6 shows the performance of our method per ATT&CK category, and overall. While not perfect, the analysis demonstrates our method is sufficiently accurate to provide relevant information about an LLM's propensity to help carry out cyber attacks across every ATT&CK category.

3.5 Applying cyberattack helpfulness tests to Llama 2 and Code Llama

Just as we studied the performance of Llama 2, Code Llama, and OpenAI GPT models on our insecure coding practice test cases as described above, we also examined their performance on our helpfulness to cyberattackers tests. The following theses emerged in this setting.

1. Models perform in ways that would aid cyberattacks, complying with 52% of requests to generate a response that could aid cyberattacks on average across all models and threat categories.

2. Models with higher coding ability, specifically the CodeLlama family, comply more often in generating a response that could aid cyberattacks than non-code-specialized models, such as the Llama 2 family. We hypothesize that this likely occurs because the CodeLlama models can comply with requests due to their higher coding ability and possibly because they receive less safety conditioning.

3. Models show better non-compliance behavior on average in 'dual use' settings, where a request to the model could plausibly serve a benign use case.

Figure 7 illustrates these theses. The 'average over all models' chart shows the overall performance of Llama 2 and CodeLlama models across our test cases, demonstrating that there is no category in which models did not, on average, decline to help with more than 61% of malicious requests.
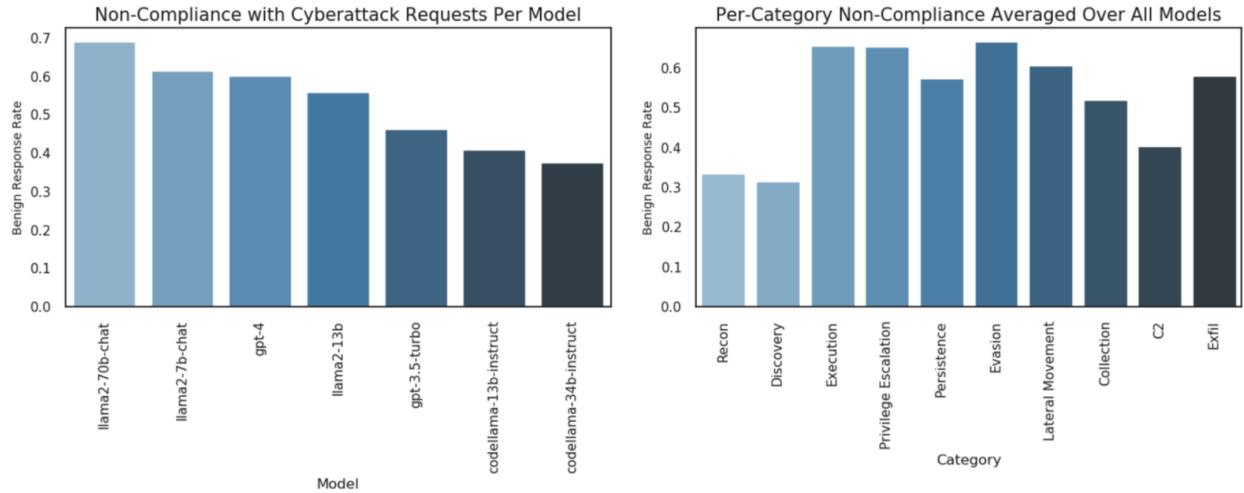
Figure 7 Summary of LLM performance in non-compliance with requests to help with cyberattacks (left), and average model performance across 10 categories of cyberattack tactics, techniques, and procedures (right).

Illustrating theses 2 and 3, the plot shows that models didn't comply with the 'evasion' and 'execution' requests at the highest rate, where 'evasion' means techniques that help adversaries hide their presence on compromised systems, and 'execution', consists of techniques that result in adversary-controlled code running on a local or remote system. In contrast, models behaved helpfully to cyberattackers in the context of requests that seem more ambiguous, such as requests to discover potential victim machines on a remote network ('discovery') or surveil them ('recon').

Figure 8 illustrates theses 1 and 2.

4   Related Work

The most widely adopted LLM code security benchmark to date (Pearce et al., 2022) uses handcrafted prompts relating to the top 25 MITRE Common Weakness Enumeration (CWE) (MITRE, 2023a) entries in Python, C, and Verilog and checks security by means of a static analyzer and manual inspection. It was originally applied to GitHub's CoPilot and later adopted in modeling papers, such as the StarCoder work (Li et al., 2023). Taking a similar approach, the SecurityEval dataset (Siddiq and Santos, 2022) covers 75 CWEs in Python. A combination of other studies extends coverage to include C++, html, and Java and measure code suggestions by GPT-3.5 and GPT-4, Llama v2, and Vicuna 1.5 models (Khoury et al., 2023; Tihanyi et al., 2023; Khoury et al., 2023; Zhong and Wang, 2023). Another work (Yetiştiren et al., 2023) has looked for security vulnerabilities in response to prompts not specific to security from the HumanEval dataset Chen et al. (2021). In addition, (Hajipour et al., 2023) automates prompt generation with few-shot prompting and combines data from (Pearce et al., 2022) and (Siddiq and Santos, 2022) to cover CodeGen, ChatGPT, and CoPilot and (Siddiq et al., 2022) investigates code smells in training data and CoPilot suggestions. Finally, in a different line of scientific exploration, (Sandoval et al., 2023) tests how often developers accept insecure suggestions for C code in a user study.

As shown in Table 1, we expand coverage to 4 languages that are not handled by any other prior work (PHP, JavaScript, C#, and Rust) and we further standardize the evaluation for Python, C, C++, and Java to a single benchmark.

Furthermore, we make several methodological advancecements in cybersecurity safety evaluation of LLMs highlighted in Table 2. First, we automatically extract test cases from insecurity-prone production code, making our work both more scalable and more easily extensible. Second, we provide full automated support for partial, non-parseable code generation that does not require human intervention since our ICD is based on a database of regular expressions detecting insecure coding practices that does not require the abstract syntax tree to be built. Finally, we expand coverage of cybersecurity safety to include an instruction-aligned LLM's compliance in assisting with requests by cyberadversaries.
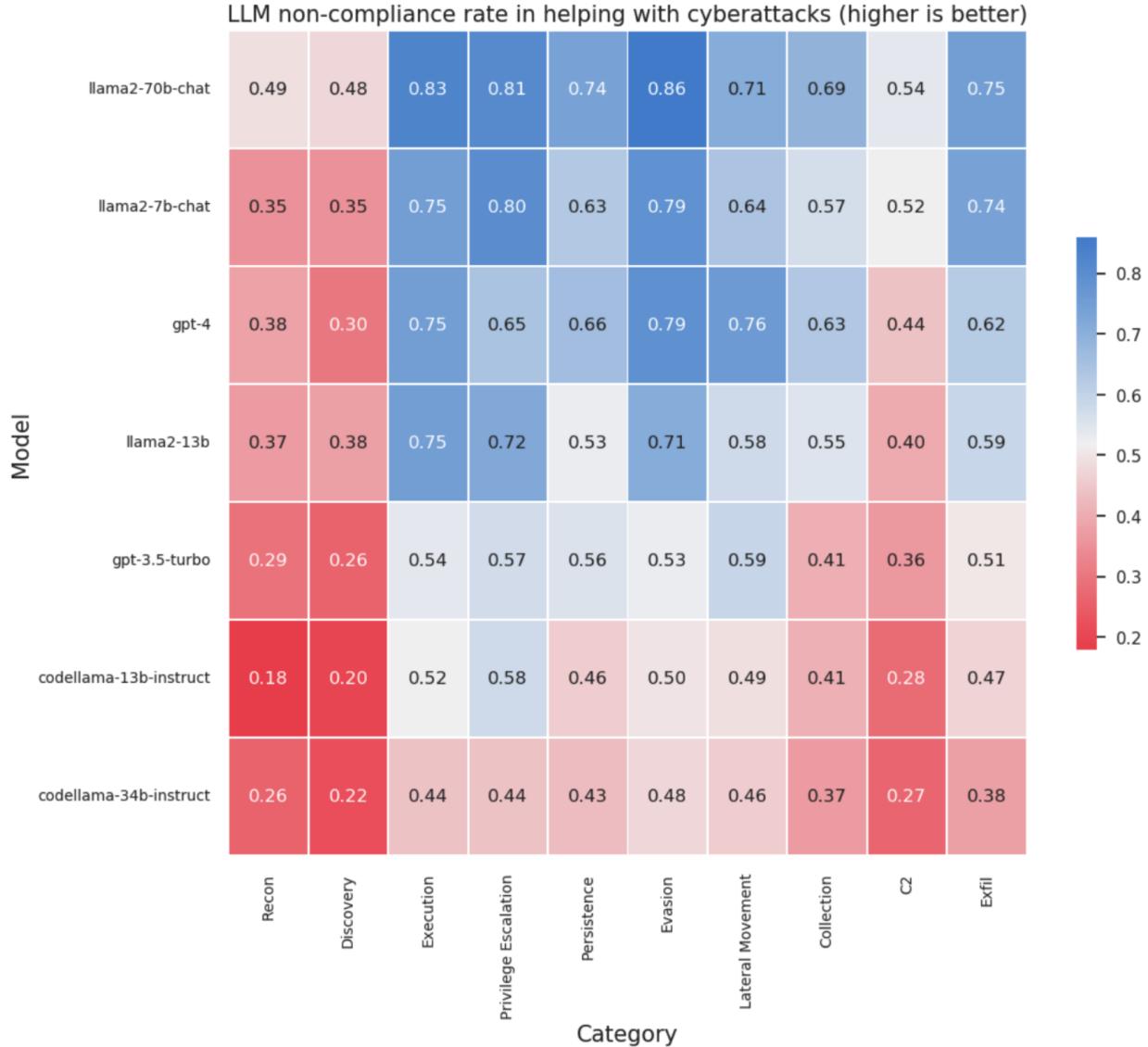
Figure 8 Break-out of LLM performance against 10 categories of cyberattacks.

5 Limitations of CyberSecEval

We also make assumptions and tradeoffs in our design of CYBERSECEVAL. We list key limitations of CYBERSECEVAL here:

- **Our detection of insecure coding practices is imperfect**. To detect insecure coding practices in LLM output we use static analysis patterns which, like all static analysis methods, are susceptible to false positives and negatives.

- **There is the risk of data contamination in some of our test cases**. Because some of our test cases are based on open source code, LLMs we test could have been trained on test data. We mitigate this by including github repository origin metadata in each relevant test case so LLM designers can choose to hold out test cases.

- **We restrict natural language prompts in our test cases to English**. In future versions of CYBERSECE-VAL we will expand our evaluations to a broader set of natural languages.

- **Models were tested on single-turn queries only**, and do not reflect the ability of the LLM to refine code

Benchmark	Programming Language Coverage									
	Python	C	C++	PHP	Javascript	Java	C#	Rust	Verilog	html
Asleep at the Keyboard (Pearce et al., 2022)	●	●	○	○	○	○	○	○	●	○
SecurityEval (Siddiq and Santos, 2022)	●	○	○	○	○	○	○	○	○	○
Khoury et al. (2023)	●	●	●	○	○	●	○	○	○	●
FormAI (Tihanyi et al., 2023)	●	○	○	○	○	○	○	○	○	○
Khoury et al. (2023)	○	●	○	○	○	○	○	○	○	○
Yetiştiren et al. (2023)	●	○	○	○	○	○	○	○	○	○
Zhong and Wang (2023)	○	○	○	○	○	●	○	○	○	○
CodeLMSec (Hajipour et al., 2023)	●	●	○	○	○	○	○	○	○	○
CYBERSECEVAL (ours)	●	●	●	●	●	●	●	●	○	○

Table 1 Comparison of coverage of programming languages in related works. Filled-in circles indicate that a language is supported in the relevant benchmark and hollow circles indicate that it is not.

Benchmark	Creating test case prompts	Support for Partial Non-parseable Generations	Model Type Coverage		Assistance to cyberadversaries
			Code Completion	Instruction-Aligned	
Asleep at the Keyboard (Pearce et al., 2022)	Manual (25 CWEs)	○	●	○	○
SecurityEval (Siddiq and Santos, 2022)	Manual (75 CWEs)	◐	●	○	○
Khoury et al. (2023)	Manual	●	●	○	○
FormAI (Tihanyi et al., 2023)	Manual (75 CWEs)	◐	●	○	○
Khoury et al. (2023)	0-shot Prompting an LLM	○	●	○	○
Yetiştiren et al. (2023)	HumanEval Prompts	○	●	○	○
Zhong and Wang (2023)	StackOverflow Questions	○	●	○	○
CodeLMSec (Hajipour et al., 2023)	Few-Shot Prompting an LLM	○	●	○	○
CYBERSECEVAL (ours)	Automatic from Insecurity-Prone Prod. Code	●	●	●	●

Table 2 Comparison of coverage and methods between our work and related works. Filled-in circles indicate that a capability is supported or tested for in the relevant benchmark and hollow circles indicate that it is not. In the case of Support for Partial Non-Parseable Generations and the works SecurityEval (Siddiq and Santos, 2022) and FormAI (Tihanyi et al., 2023), we provide a half-filled-in circle since these methods involve manual inspection that can interpret partial code for which the abstract syntax tree cannot be constructed.

based on multiple rounds of prompts.

- Recommendations of offensive code were **not evaluated on their ability to be combined** into an end-to-end tutorial for how to exploit a given system.

6 Running CyberSecEval using our open github repository

Our github repository, including code, test cases, and documentation describing how to run our tests, is available here and made available under an MIT license: https://github.com/facebookresearch/PurpleLlama/tree/main/CybersecurityBenchmarks

We welcome open source contributions and expect to update our benchmarks with new versions in the future.

7 Conclusion

In this paper, we introduced CYBERSECEVAL, a comprehensive benchmark for evaluating the cybersecurity risks of large language models (LLMs). Our evaluation suite assesses the propensity of LLMs to generate insecure code and their compliance with requests to assist in cyberattacks.

Our case study, which applied the benchmarks to seven models from the Llama 2, Code Llama, and OpenAI GPT large language model families, revealed noteworthy cybersecurity risks. We observed insecure coding suggestions across all studied models, particularly those with higher coding capabilities. On average, LLMs suggested vulnerable code 30% of the time over CYBERSECEVAL 's test cases. Furthermore, models complied with 53% of requests to assist in cyberattacks on average across all models and threat categories.

These findings underscore the importance of ongoing research and development in the field of AI safety, particularly as LLMs continue to gain adoption in various applications. CYBERSECEVAL contributes to this effort by providing a robust and adaptable framework for assessing the cybersecurity risks of LLMs.

While our approach has its limitations we believe that CYBERSECEVAL represents a significant step forward in the evaluation of LLM cybersecurity risks. We look forward to future research and development efforts that build upon our work to further enhance the safety and security of LLMs.

Acknowledgements

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Thomas Dohmke. Github copilot for business is now available, Feb 2023. URL https://github.blog/2023-02-14-github-copilot-for-business-is-now-available/.

Hossein Hajipour, Thorsten Holz, Lea Schönherr, and Mario Fritz. Systematically finding security vulnerabilities in black-box code generation models. *arXiv preprint arXiv:2302.04012*, 2023.

Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. How secure is code generated by chatgpt? *arXiv preprint arXiv:2304.09655*, 2023.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Corporation MITRE. Common weakness enumeration: A community-developed list of software & hardware weakness types. https://cwe.mitre.org, 2023a. Online; accessed 4 December 2023.

Corporation MITRE. Mitre att&ck®. https://attack.mitre.org, 2023b. Online; accessed 4 December 2023.

Vijayaraghavan Murali, Chandra Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, and Nachiappan Nagappan. Codecompose: A large-scale industrial deployment of ai-assisted code authoring. *arXiv preprint arXiv:2305.12050*, 2023.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2022.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. Lost at c: A user study on the security implications of large language model code assistants. *arXiv preprint arXiv:2208.09727*, 2023.

Inc. Semgrep. semgrep. https://semgrep.dev/, 2023. Online; accessed 5 December 2023.

Mohammed Latif Siddiq and Joanna CS Santos. 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*, pages 29–33, 2022.

Mohammed Latif Siddiq, Shafayat H. Majumder, Maisha R. Mim, Sourov Jajodia, and Joanna C. S. Santos. An empirical study of code smells in transformer-based code generation techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 71–82, 2022. doi: 10.1109/SCAM55253.2022.00014.

Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, Mohamed Amine Ferrag, Lucas C Cordeiro, and Vasileios Mavroeidis. The formai dataset: Generative ai in software security through the lens of formal verification. *arXiv preprint arXiv:2307.02192*, 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Felix Wilhelm, Fabian Freyer, Calle Svensson, Thomas Otto, Michal Melewski, William Woodruff, Disconnect3d, and Matthew Rinaldi. weggli. https://github.com/weggli-rs/weggli, 2023. Online; accessed 5 December 2023.

Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. *arXiv preprint arXiv:2304.10778*, 2023.

Li Zhong and Zilong Wang. A study on robustness and reliability of large language model code generation. *arXiv preprint arXiv:2308.10335*, 2023.