Pairing Security Advisories with Vulnerable Functions Using Open-Source LLMs

Trevor Dunlap[1,2], John Speed Meyers[2], Bradley Reaves[1], and William Enck[1]

[1] North Carolina State University, Raleigh, NC, USA
[2] Chainguard, Kirkland, WA, USA
{trevor.dunlap, jsmeyers}@chainguard.dev
{bgreaves, whenck}@ncsu.edu

Abstract. As the reliance on open-source software dependencies increases, managing the security vulnerabilities in these dependencies becomes complex. State-of-the-art industry tools use reachability analysis of code to alert developers when security vulnerabilities in dependencies are likely to impact their projects. These tools heavily rely on precisely identifying the location of the vulnerability within the dependency, specifically vulnerable functions. However, the process of identifying vulnerable functions is currently either manual or uses a naive automated approach that falsely assumes all changed functions in a security patch link are vulnerable. In this paper, we explore using open-source large language models (LLMs) to improve pairing security advisories with vulnerable functions. We explore various prompting strategies, learning paradigms (i.e., zero-shot vs. few-shot), and show our approach generalizes to other open-source LLMs. Compared to the naive automated approach, we show a 173% increase in precision while only having an 18% decrease in recall. The significant increase in precision to enhance vulnerable function identification lays the groundwork for downstream techniques that depend on this critical information for security analysis and threat mitigation.

Keywords: Vulnerable Function · Security Advisory · Security Database · Large Language Model

1 Introduction

In software development, leveraging open-source dependencies is fundamental, accelerating innovation and development processes. Yet, these dependencies often carry the burden of security vulnerabilities, posing significant management challenges [2]. This challenge has led to the widespread use of software composition analysis tools (e.g., OWASP Dependency-Check, GitHub Dependabot) that notify developers of potential vulnerabilities in their dependencies. One shortcoming is that these tools often overgeneralize by flagging any inclusion or use of a known vulnerable dependency as potentially vulnerable, resulting in a high rate of false positives [10] and alert fatigue [1].

State-of-the-art industry tools [28,14,17] have begun to determine how those dependencies are used within projects. These tools rely on a program analysis concept known as *reachability* to determine whether functions containing vulnerable portions of code are ever called within dependencies. By assessing if vulnerable functions are called, developers can prioritize genuine threats, focusing on vulnerabilities that pose an actual risk to their projects. However, the success of reachability analysis first depends on accurately identifying vulnerable functions within dependencies. Incorrectly identifying safe functions as vulnerable causes unnecessary alerts for developers about threats that do not exist.

There are two common approaches for pairing security advisories with vulnerable functions: (1) manual curation and (2) using a naive automated approach that considers all changed functions in a security patch link as vulnerable. The Go Vulnerability Database (GoVulnDB [16]), supported by Google's Go security team, exemplifies the manual approach by releasing high-quality security reports for the Go community. In these reports, the Go security team manually identifies the root cause of vulnerabilities, accurately pinpointing the exact vulnerable functions. Despite its high accuracy and the trust it builds, this manual process is time-consuming and limited in scalability.

Alternatively, the automated approach, often seen in academic literature [40,7,15,27,4], is intuitive but flawed. The automated approach assumes all modified functions in a security patch are vulnerable. The underlying assumption is that developers are only addressing the vulnerability and not any additional changes during the patching process. However, our preliminary analysis found that out of all the functions the automated approach labeled as vulnerable, only 22% were correct when using GoVulnDB as ground truth. This means that 78% of the functions modified in a security patch were unrelated to the actual vulnerability, underscoring the significant issue of overestimation and highlighting the need for improved automated methods.

In this paper, we hypothesize that LLMs can significantly improve the accuracy of pairing security advisories with vulnerable functions over the current naive automated approach. Our high-level approach is to take a vulnerability description from the security advisory and the associated changes from the security patch link to ask the model if the changes fix the underlying vulnerability. We explore various model sizes (i.e., CodeLlama 7 billion (B) parameters, 13B, and 34B versions [31]) using a variety of prompting strategies. We develop three prompts: a simple standard prompt, a detailed prompt, and a chain-of-thought prompt that involves the model providing an explanation [38]. We evaluate these prompts in two learning paradigms: either zero-shot or few-shot. The zero-shot approach does not have examples in the prompt, and the few-shot relies on our retrieval system to obtain similar examples to provide in the prompt to help guide the model. Additionally, we evaluate the computation times for each strategy. Finally, we explore how these techniques generalize to three other popular code-oriented LLMs (Mixtral [21], WizardCoder [26], and DeepSeek [26]).

Our experimentation demonstrates the following key findings:

- LLMs surpass the naive automated approach that assumes all functions modified in a security patch are vulnerable. We show a 173% increase in

```
func unzipFile(file *zip.File, dstDir
     string) error {
-  filePath := path.Join(dstDir, file
     .Name)
+  name := strings.TrimPrefix(
     filepath.Join(string(filepath.
     Separator), file.Name), string(
     filepath.Separator))
+  filePath := path.Join(dstDir, name
     )
```

(a) Modifications in the *unzipFile* function prevent a path traversal vulnerability by sanitizing the file name before joining the destination directory path.

```
func RemoveFile(path string) error {
-  err := os.Remove(path)
-  return err
+  return os.Remove(path)
}
```

(b) Refactoring of the *RemoveFile* non-vulnerable function to directly return the result of os.Remove(path), removing the temporary variable.

Fig. 1: Condensed updates of multiple functions in the patch for the Go utility library, *go-huge-util*, addressing CVE-2023-28105. Figure 1a are changes to the root causing vulnerable function, *unzipFile*. Figure 1b shows unrelated changes to the underlying vulnerability in function *RemoveFile*.

precision (i.e., from 0.22 to 0.60) with only an 18% decrease in recall (i.e., from 0.90 to 0.74).

- Smaller models, with appropriate prompting, can match or even exceed the performance of larger models, suggesting a resource-efficient alternative.
- While chain-of-thought prompting proves the best approach in zero-shot settings, it comes at a 13x slowdown compared to more streamlined prompting with a few-shot setting that can show similar results.
- Our approach generalizes to multiple LLMs, allowing downstream users to pair security advisories with vulnerable functions.

We have released our data and implementation on GitHub.[3]

2 Background

2.1 Motivating Example

Security advisory CVE-2023-28105 highlights a path traversal vulnerability in the *go-huge-util* Go library. Within the security advisory, a reference link exists, also known as a patch link, that points to how a developer applies a fix for the vulnerability. These patch links are the basis for identifying vulnerable functions. However, the patch for CVE-2023-28105 does more than fix the vulnerable function, as described by the developer in the commit message: *"fix zip.Unzip path traversal vulnerability and add some new file utility functions."* Specifically, the patch has modifications in four files, modifies five existing functions, adds two new functions, and totals 104 line additions and 45 line deletions. The comprehensive

[3] https://github.com/s3c2/llm-vulnerable-functions

```
id: GO-2023-1640
modules:
    - module: github.com/dablelv/go-huge-util
      packages:
        - package: github.com/dablelv/go-huge-util/zip
          symbols:
            - unzipFile
          derived_symbols:
            - Unzip
cves:
    - CVE-2023-28105
references:
    - fix: https://github.com/dablelv/go-huge-util/commit/0e308b0
```

Fig. 2: Condensed GO-2023-1640 report for CVE-2023-28105.

changes within the patch link underscore the complexity of accurately identifying vulnerable functions.

Figure 1a contains the changes to the root causing vulnerability within the patch link for CVE-2023-28105 and its fix in the *unzipFile* function. The fix was straightforward: the file path input requires sanitization to prevent the traversal vulnerability. Interestingly, the security advisory and developer mentions the function *zip.Unzip*: *"When users use 'zip.Unzip' to unzip zip files from a malicious attacker, they may be vulnerable to path traversal."* However, the *Unzip* function was unchanged during the patch link, and this is simply the public-facing function that calls the underlying vulnerable function, *unzipFile*.

In addition to fixing the underlying path traversal vulnerability, other changes occurred within the patch link. Figure 1b is an example of one of the five existing functions modified during the patch link unrelated to fixing the path traversal vulnerability. In this case, the developer refactored the function *RemoveFile* to directly return the status of removing the given file instead of saving the result in a temporary variable. Using a naive approach to label all changed functions in a patch link as vulnerable would misidentify the non-vulnerable function *RemoveFile* as vulnerable, thus creating noise for downstream data consumers.

2.2 Go Vulnerability Database (GoVulnDB)

The Go Vulnerability Database (GoVulnDB), supported by Google, is a high-quality dataset of vulnerabilities in Go [16]. The Go Security team compiles data from various datasets (e.g., the National Vulnerability Database, GitHub Advisory Database). The Go Security team then manually checks reports for precise details about the vulnerabilities, including descriptions and affected versions [18].[4]

Vulnerable Functions: One unique aspect of GoVulnDB is its curation of vulnerable functions. These vulnerable functions are in two distinct fields: *symbols* and *derived symbols*. The *symbols* field identifies the root causing vulnerable functions, such as functions or methods, that directly cause the reported

[4] After the submission of this paper, GoVulnDB started experimenting with assigning any changed functions within the patch links as vulnerable.

Algorithm 1: Generalization of the naive automated approach used in prior work [40,7,15,27,4].

```
Input: $patchLink$ from the security advisory
Output: $functions$ unique set of changed functions in the patch
$ChangedFuncs(patchLink)$:
    $functions \leftarrow$ empty set
    for $(gitHunk, fileName) \in PatchParser(patchLink)$ do
        if "test" not in ToLower($fileName$) then
            for $lineNumber \in gitHunk$ do
                for $func \in$ Tree-Sitter($fileName$) do
                    if $func.startLine \leq lineNumber \leq func.endLine$ then
                        $functions$.add($func.name$)
    return $functions$
```

vulnerability. These initial symbols are *manually* identified by the Go security team. The *derived symbols* represents a further analysis of public functions that reach the root causing vulnerable symbols. These *derived symbols* are generated automatically using portions of their static analysis tool govulncheck [17] and converted into reports using their vulnreport tool [19]. The GoVulnDB is a reliable, ground-truth foundation for understanding the nature of pairing security advisories with their associated vulnerable functions.

Figure 2 shows the GoVulnDB report, GO-2023-1640, for the prior motivating examples. The Go security team manually reviewed the report and identified the root causing vulnerable function as *unzipFile*. When automatically generating the *derived symbols*, the *Unzip* function is identified as the public-facing function initially mentioned in the security advisory description.

3 Naive Automated Approach (*ChangedFuncs*)

While manual curation of vulnerable functions delivers high-quality data (e.g., GoVulnDB), it suffers in scalability. The alternative approach, commonly seen in academic literature [40,7,15,27,4], uses an automated approach. The automated approach, while intuitive, is naive. The underlying assumption is that all modified functions within a patch link are vulnerable. However, we find the approach only correctly identifies 22% of the root causing vulnerable functions (Section 5.2), further emphasizing the need for an accurate automated approach.

To replicate the naive automated approach for identifying vulnerable functions, we implement Algorithm 1, referred to as *ChangedFuncs*. This process begins with a patch link from a security advisory. Such patch links lead to specific commits in a project's source code repository, highlighting the changes. These commits, also known as git-diffs, appear in *hunk* levels (git-hunks). A git-hunk represents a block of code changes within a file. Naturally, git-hunks do not specify if these changes are within functions, as modifications can occur inside and outside functions. Determining if the git-hunks are modifying code inside

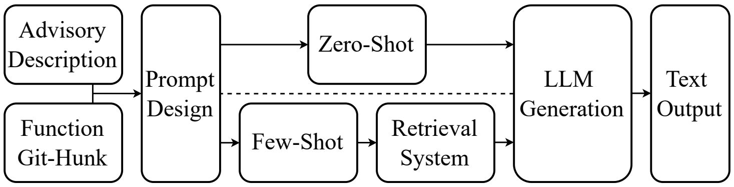


Fig. 3: Our design approach involves using either a zero-shot or few-shot technique for pairing security advisories with vulnerable functions.

of a function is trivial. We first parse the patch links with PatchParser [13], a Python library that transforms commits into machine-readable data and provides the associated line numbers of git-hunk modifications. Using Tree-sitter [33], a language-agnostic parsing library, we identify the functions' start and end line numbers within the same file the git-hunk appears in. If a git-hunk's modifications are within a function's line range, the git-hunk is labeled with the function name. We also exclude files containing the word "test," ensuring only non-test functions impacted by the patch are considered. In our results, we use *ChangedFuncs* as a baseline, treating each identified function within the patch link as vulnerable.

4 LLM Approach

Figure 3 overviews our approach for using Large Language Models (LLMs) to pair security advisories with vulnerable functions. Our approach consumes a given security advisory (i.e., GoVulnDB report) and individual function code changes (i.e., git-hunks) from the associated patch link within the security advisory. We design three prompts: a standard, detailed, or chain-of-thought prompt, all aimed at determining whether the code changes address the vulnerabilities described in a security advisory. We compare these prompts in two scenarios: a "zero-shot" setting, where the LLM operates without prior examples, and a "few-shot" setting, incorporating similar examples into the prompt to guide the LLM's analysis. These similar examples rely on our retrieval system. From there, the prompt is given to the LLM to generate an output to determine if the given function changes are related to fixing the specific security advisory description.

4.1 Input Data

Our approach relies on two inputs: the description from the security advisory and associated function changes (i.e., git-hunk) from the patch link.

Security Advisory Description: We rely on the free-form text description given within the security advisory. We preprocess the description details by removing any additional spaces and new lines.

Function Git-Hunk: The input granularity within the system is at the git-hunk level based on the patch links provided within the security advisory. We chose this level as it helps separate individual changes for a commit.

4.2 System Prompt Design

LLMs rely on system prompts to instruct the model on what to do. We design three system prompts to instruct the model to determine if the git-hunk is responsible for fixing a given security advisory. 1) The standard prompt, designed to be straightforward, asks if the provided git-hunk fixes a vulnerability. 2) The detailed prompt has more specific instructions for the model, asking to consider the git-hunk's direct relevance to the vulnerability description. These prompts instruct the model to respond in a boolean fashion of either *True* or *False*. Where *True* represents the git-hunk was indeed fixing or related to the vulnerability description, and *False* would mean the fix is unrelated. 3) The third prompt uses the chain-of-thought method [38], prompting the model to explain the git-hunk's function before producing a decision. Each of our prompts is as follows:

> ***Standard Prompt:*** I want you to act as a vulnerability fix detection system. Determine if the following git-hunk fixed a vulnerability. Respond with only 'True' or 'False.'

> ***Detailed Prompt:*** Your task is to analyze the provided code changes (GIT-HUNK) to determine if they target a specific vulnerability (Fixed Vulnerability Description). Review each line, considering its direct relevance to the vulnerability description. Ignore any new vulnerabilities that may have been introduced. Answer 'True' if changes are directly related to fixing or even somewhat partially related to fixing the vulnerability. Only provide a 'False' conclusion if the GIT-HUNK changes are absolutely unrelated to the vulnerability. Respond with only 'True' or 'False.'

> ***Chain-of-Thought (CoT):*** Your task is to analyze the provided code changes (GIT-HUNK) to determine if they target a specific vulnerability (Fixed Vulnerability Description). Review each line, considering its direct relevance to the vulnerability description. Ignore any new vulnerabilities that may have been introduced. Provide a brief description of what the GIT-HUNK changes have done, then conclude by labeling 'True' if changes are directly related to fixing or even somewhat partially related to fixing the vulnerability. Only provide a 'False' conclusion if the GIT-HUNK changes are absolutely unrelated to the vulnerability. Justify your decision before ending with a clear 'True' or 'False' decision. Do not answer right away. Answer in the following format:
> Explanation - {Your Explanation}
> Final Decision - {True/False}

4.3 Zero-shot and Few-shot Learning Paradigms

Two distinct learning paradigms exist in LLMs: zero-shot and few-shot learning. The primary difference between the two is if example data is present in the prompt. Figure 4 illustrates the high-level template for each approach. A primary

```
{system prompt}

# System Input #
Advisory Description:
{advisory-description}
Function GIT-HUNK:
=====================
{git-hunk}
=====================
```

(a) Zero-shot prompt template for pairing security advisories to vulnerable functions.

```
{system prompt}

{retrieval system examples}

# System Input #
Advisory Description:
{advisory-description}
Function GIT-HUNK:
=====================
{git-hunk}
=====================
```

(b) Few-shot prompt, note the addition of the retrieval-based examples.

Fig. 4: Simplified prompt templates for both the zero-shot and few-shot paradagms. The difference is the addition of the retrieval examples placed after the system prompt within the few-shot template.

benefit of these approaches is that they do not require end-users to fine-tune any models and avoid the significant effort of obtaining training data.

Zero-Shot: The zero-shot learning paradigm prompts an LLM without examples [30]. Instruction-tuned models, such as the ones used within our experiments, have shown promising capabilities in zero-shot learning [37]. Additionally, zero-shot is useful when end-users have little to no example data.

Few-Shot: Contrary to zero-shot, few-shot learning incorporates a set of examples within the prompt [6]. These examples serve as a guide, allowing the LLM to grasp the context and specifics of the new task quickly. This approach is beneficial in limited data scenarios, but a set of known ground truth labels must exist. In the few-shot setting, these examples are added directly after the system prompts, shown in Figure 4b. However, determining which examples to provide within the prompt is essential for few-shot learning.

4.4 Retrieval System

The retrieval system enhances few-shot learning by providing relevant examples to the git-hunk under analysis. The relevant examples are returned through a similarity comparison process of the known labeled examples and the git-hunk in question. We begin by building an offline datastore of known labeled examples, with the GoVulnDB serving as our primary data source (as detailed in Section 5.1). The core of this system lies in transforming these git-hunks into embeddings, which are compact vector representations that encapsulate their semantic meaning. These embeddings are typically generated with a specialized code embedding model (i.e., CodeT5+ [36]). The embeddings are then stored in a datastore optimized for vector retrieval (i.e., Faiss index [12]). Upon querying,

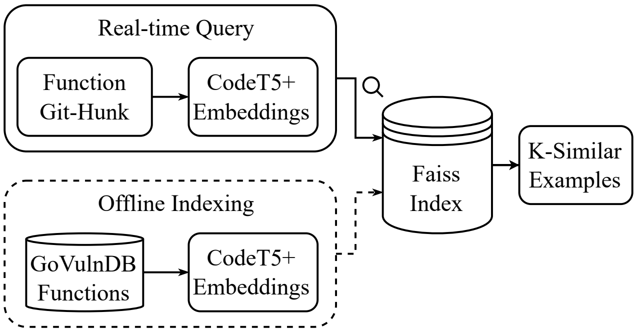


Fig. 5: The retrieval system takes the function git-hunk as input. Then, it returns k-similar examples based on the set of labeled GoVulnDB changed functions at the git-hunk level from the ground truth data.

the system fetches both *True* and *False* examples of git-hunks with the associated advisory description similar to the git-hunk under analysis.

Populating the Retrieval System with Examples: To populate the retrieval system, we rely on a model specifically designed to create embeddings, the CodeT5+ 110M embedding model [36]. Wang et al. [36] extensively evaluated CodeT5+ and found it performed exceptionally well for retrieval tasks. The input into the embedding model is at the git-hunk level. The resulting embeddings from this process are 256-dimensional vectors that we store within a Faiss index [12].

Querying the Retrieval System: When inputs come into the retrieval system, we use the same CodeT5+ embedding model as previously described. When querying the retrieval system, the Faiss index computes a similarity metric (based on the dot product) between the input vector and all pre-populated vectors. The search returns the top-K similar git-hunks with the associated advisory description of both *True* and *False* examples in alternating order. Leading with the most similar *True* example first, then the next most similar *False* example. We show the impact of this ordering in Section 5.4. We also ensure that the examples come from a different advisory and git-hunk, confirming that we are not leaking data or labels at inference time. The overall retrieval system is implemented using the LangChain library [23].

4.5 LLM Text Generation

LLMs use an autoregressive generation method, where text is generated iteratively by predicting the next words, more specifically tokens, based on the context of all previously generated tokens [34]. This process involves a sequence of forward passes through the model's network, each pass calculating the probability distribution for the next possible token. The next token is selected based on this distribution and then appended to the growing text sequence. Generating longer sequences increases computational time due to the number of iterative passes required through the model. Consequently, the output format from the LLM (e.g., boolean responses vs. an explanation) significantly impacts generation time.

Standard and Detailed Prompt Responses: When we use the standard and detailed prompts, we aim for a boolean single-word response of *True* or *False* from the model. This requires just a single forward pass through the model to produce the boolean response. At the end of this step, the model produces preliminary scores, known as logits, for each potential token in its vocabulary [3]. These logits are the model's initial guesses, not yet probabilities. We then use a function called softmax to turn these logits into actual probabilities, showing how confident the model is in each outcome. By focusing on the probabilities for *True* and *False* tokens, we can identify the model's decision.

Chain-of-Thought Responses: The expected result is a longer free-form text when using the chain-of-thought (CoT) prompt. This is because the model explains what the git-hunk has changed, then concludes with a definitive *True* or *False*, categorizing the git-hunk as related or unrelated to the vulnerability fix. A consequence of the longer text will be an increase in computation time. To identify the final decision, we search the text for either *True* or *False* values.

4.6 Aggregating Results to the Function Level

Predictions from the LLM are at the git-hunk level. However, this paper aims to associate security advisories with entire functions rather than individual git-hunks. In some cases, multiple changes can occur within a single function spanning multiple git-hunks, requiring us to aggregate the results into one function. To do so, if any git-hunk of a function is predicted as *True*, indicating a relation to the vulnerability, we classify the entire function as vulnerable. This methodology assumes that the vulnerability impacts the whole function if any portion of the function is involved in the vulnerability fix.

5 Results

We evaluate the use of LLMs to pair security advisories with vulnerable functions with the goal of addressing the following questions:

Q1: How well does the naive approach work for identifying vulnerable functions?
Q2: How do various prompts impact the performance of using LLMs?
Q3: How does zero-shot and few-shot learning impact the effectiveness of LLMs?
Q4: How does the computational time compare across prompt strategies?
Q5: Can the developed prompts generalize from one LLM to another LLM?

Key Takeaways: Our analysis shows that using open-source large language models (LLMs) to link security advisories with vulnerable functions presents various insights and challenges. Current automated detection methods, like the naive *ChangedFuncs* approach, identify only 22% of actual vulnerable functions, leading to many false positives. This highlights the need for more advanced techniques. LLMs can improve precision in detecting these functions by up to 173% compared to *ChangedFuncs*; however, they have an 18% drop in recall. The study also finds that the effectiveness of LLMs varies with the quality of the input prompt, with well-designed prompts significantly boosting the performance of smaller models. Few-shot learning enhances recall but comes at the cost of reducing precision. The Chain-of-Thought prompting method excels in performance but requires significantly more computational resources, 13 times slower than other methods. Finally, our findings apply across different LLMs.

5.1 Experimental Setup

Throughout the results, we primarily focus on the CodeLlama family of models [31]. The CodeLlama family is designed explicitly for coding tasks and has instruction-following versions, each available in 7 billion (B), 13B, and 34B parameter configurations. Additionally, we also evaluate other popular coding-oriented LLMs: Mixtral 7x8B [21], DeepSeek [5], and WizardCoder [26]. We use a machine with an Intel i7-9700k CPU, 128GB RAM, and two NVIDIA RTX 3090 Ti GPUs. During evaluation, each model is split across both GPUs with 4-bit quantization and parameters set to half-precision (float16). We use the same CodeLama parameters as Meta for text generation throughout [31]. We evaluate against the GoVulnDB data described in Section 5.1.

Data Collection: We rely on the GoVulnDB as ground truth data. We initially cloned GoVulnDB on October 26, 2023, and obtained 409 GoVulnDB reports. Out of these, 318 (78%) had an associated patch link. Of those 318 security advisories, 280 (88%) of the reports listed vulnerable functions. When a vulnerable function is not listed, the entire project is considered vulnerable; therefore, we do not consider those for further analysis. These 280 reports form the foundation for our subsequent analysis. Using *ChangedFuncs*, Algorithm 1, we identify 2,370 modified functions from 298 patch links across the 280 GoVulnDB reports. From GoVulnDB, 528 of those functions were labeled as vulnerable functions. We consider the remaining 1,842 functions non-vulnerable. We note that some git-hunks can change multiple functions; we split these git-hunks to ensure that only one function is processed at a time.

5.2 Comparing the *ChangedFuncs* to GoVulnDB

When comparing *ChangedFuncs* (Section 3) to the GoVulnDB dataset, we found the approach produces a precision of 0.22. The remaining functions (1,842 functions) changed during the patch links were not listed in the GoVulnDB reports, resulting in false positives. Furthermore, 61 reported vulnerable functions were not found within the patch link. The overall recall rate is 0.9, demonstrating the ability to capture the majority of vulnerable functions. Through a manual analysis, we found that the missing functions were either due to a missing patch link from the advisory or were mentioned in a subsequent link (e.g., a reference to the Issue). Researchers and practitioners using the naive automated approach will create significant noise for downstream consumers.

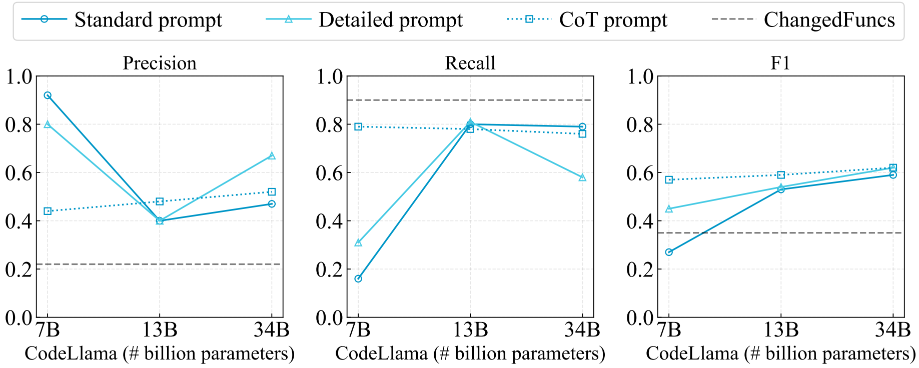


Fig. 6: Zero-shot setting between the standard, detailed, and chain-of-thought (CoT) prompt across CodeLlama 7B, 13B, and 34B. Prompt design has a greater positive impact on the smaller 7B model, allowing it to produce similar results to the larger 13B and 34B models.

5.3 LLM Prompt Impacts

Figure 6 compares the standard, detailed, and CoT prompts from Section 4.2 using the CodeLlama model family (7B, 13B, 34B) in a zero-shot context. The *ChangedFuncs* approach, assuming all changed functions in a patch link are vulnerable, serves as the baseline. The evaluation focuses on *True* labeled functions to ensure fairness against the *ChangedFuncs* baseline.

The prompts have the largest impact on the smaller 7B model. For instance, switching from the standard to the detailed prompt resulted in a 66.7% increase (0.27 to 0.45) in the F1 score. Then, switching from the detailed prompt to the CoT prompt resulted in another 26.7% increase (0.45 to 0.57) in the F1 score. However, these increases in F1 have tradeoffs in precision and recall. Both the detailed and CoT decreased the precision and increased the recall in the 7B model. A higher precision and lower recall indicate the model was predominately responding *False* to each input, with fewer *True* responses. However, when the recall increased, the model responded with more *True* responses.

The prompts have a less significant impact on the larger CodeLlama 13B and 34B models. On average, the 13B version has a 5.6% increase (ranging from 0.53 to 0.59) in F1 score across the prompts, and the 34B version has a 2.5% increase (ranging from 0.59 to 0.62). For the larger CodeLlama-13B model, the prompts have similar performance across precision and recall. The largest CodeLlama-34B model performed similarly using the standard and CoT prompts. However, the detailed prompt has a higher precision of 0.67 and a lower recall of 0.58. Both the detailed and CoT prompt in the CodeLlama-34B model had the highest F1 score of 0.62. Overall, the prompts have a minor impact on the larger models.

An additional benefit of the CoT prompt is the extra explanation provided. Below is an example of the response from the CodeLlama-34B zero-shot setting from Figure 1a in the Background:

Table 1: Zero-shot vs Few-Shot in terms of F1 across CodeLlama (7B, 13B, 34B) using the standard, detailed, and CoT prompt. Best metrics are bolded.

	CodeLlama-7B				CodeLlama-13B				CodeLlama-34B			
# Examples	0	3	5	10	0	3	5	10	0	3	5	10
Standard	0.27	0.38	**0.39**	0.35	0.53	**0.57**	0.54	0.53	0.59	0.53	0.54	**0.60**
Detailed	0.45	0.53	**0.58**	0.52	0.54	**0.56**	0.56	0.54	**0.62**	0.57	0.59	**0.62**
CoT	0.57	0.56	0.59	**0.60**	0.59	**0.60**	**0.60**	0.55	0.62	**0.63**	0.62	0.61

> The GIT-DIFF changes have addressed the path traversal vulnerability in the ‘unzipFile‘ function by using the ‘path.Join‘ and ‘strings.TrimPrefix‘ functions to ensure that the file path is valid and does not contain any malicious traversal sequences. Final Decision: True

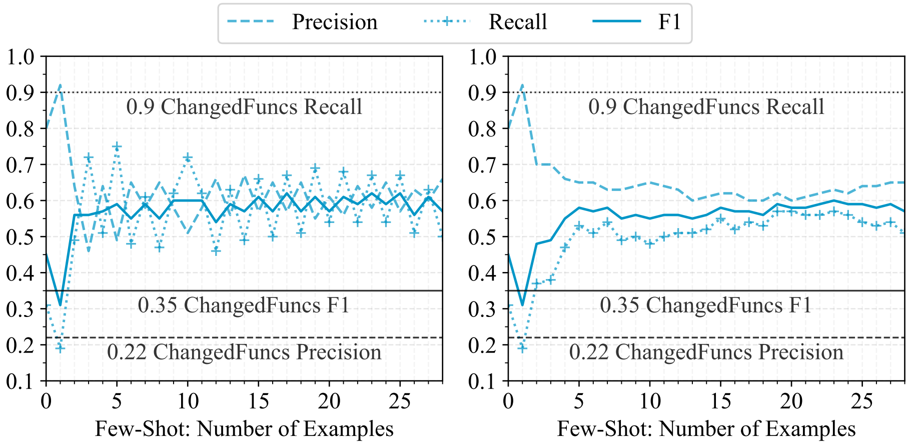
5.4 Zero-Shot vs. Few-Shot

Table 1 compares the performance of the CodeLLama 7B, 13B, and the 34B model in a few-shot learning context using the standard, detailed, and CoT prompting. The few-shot approach helps improve performance compared to the zero-shot approach, primarily in recall in the basic and detailed prompt settings.

Introducing a few-shot paradigm has the largest impact on the smaller CodeLlama 7B model. On average, each prompt in a few-shot setting increased the F1 score of the CodeLlama 7B model by 26.2%. Few-shot learning increased the CodeLlama 13B version's F1 on average by 4.1%, and the CodeLlama 34B version's by only 1.1%. The overall trend for the standard and detailed prompt after adding few-shot examples was an increase in recall and a decrease in precision. The CoT prompt had the opposite reaction, typically seeing a minor decrease in recall and an increase in precision. Interestingly, a few-shot approach in the detailed and CoT prompts allows the smaller CodeLlama 7B model to perform similarly to the 13B and 34B versions.

Example Order: When experimenting with the few-shot paradigm, we noticed that the order in which the examples in the prompt appear impacts the model performance, as shown in Figure 7. For instance, our retrieval system returns both *True* and *False* examples (detailed in Section 4.4). Once examples are returned, we place them in an alternating order in the prompt: start with a *True* example followed by a *False* example.

Figure 7a shows the alternating impact of precision and recall depending on the number of examples in the prompt. The number of examples changes the type of example before the input. For instance, an even number of examples ends with a *False* example, and an odd number ends with a *True* example. A closing *True* example tends to influence the model towards a true prediction, enhancing recall but at the cost of precision. While a closing *False* example influences the model towards false predictions, improving precision while reducing recall. In Figure 7b, we randomly shuffle the order of examples to counteract the observed



(a) Examples Ordered by Similarity (b) Examples Order Randomly Shuffled

Fig. 7: Randomly shuffling the examples in a few-shot setting of CodeLlama 7B using the detailed prompt stabilizes the overall performance of the model.

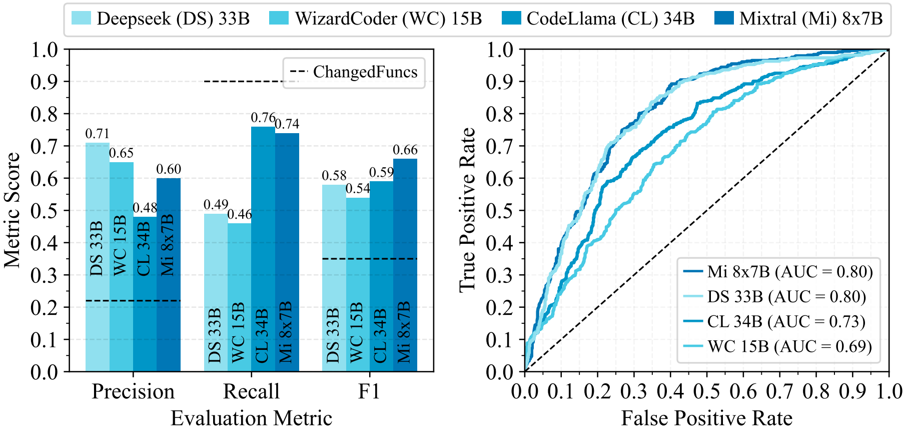
bias. Our findings suggest that structured example sequencing creates tradeoffs between precision-recall, whereas randomization offers stable model performance.

Retrieval Impact: Additionally, we observed the similarity of returned examples impacts the model's predictive performance. Specifically, a higher mean retrieval score for a given example type (e.g., *True* examples) correlates with the model predicting that the target function addressed a vulnerability. A greater discrepancy in retrieval scores between *True* and *False* examples led to more accurate predictions by the model. However, when retrieval scores for *True* and *False* examples were similar, the model had difficulty distinguishing between the two, often mislabeling functions as *True*. This pattern was also evident in missed instances, where *False* examples had higher overall retrieval scores. The influence of retrieval scores on model accuracy was greater under an alternating example approach, though the trend continues with randomly shuffled examples.

5.5 Generalization across additional LLMs

Here we demonstrate the performance of prompts across the Mixtral 8x7B [21], DeepSeek 33B [5], WizardCoder 15B [26], and CodeLlama 34B [31] models. We evaluate using the 5-shot detailed prompt for each model.

Figure 8a specifically focuses on performance regarding *True* labels. Notably, Mixtral 8x7B has the highest overall F1 score. Specifically, its precision is a 173% improvement over the *ChangedFuncs* (0.60 vs. 0.22) and an 18% decrease in recall (0.90 vs 0.74). In contrast, CodeLlama 34B has a slightly higher recall than Mixtral 8x7B but significantly reduces precision (0.60 to 0.48). DeepSeek and WizardCoder achieved higher precision scores but at the expense of lower



(a) *True* label evaluation metrics

(b) ROC curve for various models

Fig. 8: Performance metrics for DeepSeek 33B, WizardCoder 15B, CodeLlama 34B, and Mixtral 8x7B using the detailed prompt in a 5-shot setting. The best overall performing model was Mixtral 8x7B, with a 173% increase in precision and an 18% decrease in recall compared to the naive approach.

recall rates than CodeLlama and Mixtral. This suggests that DeepSeek and WizardCoder were more prone to producing *False* responses than the other two models. We note that the Mixtral 8x7B is the largest model (47B parameters [21]) of any code model we evaluated, which could be a factor in the higher performance.

Figure 8b focuses on the models' ability to distinguish *True* from *False* labels. The ROC curve is based on the probabilities from the *True* and *False* tokens as described in Section 4.5. Mixtral 8x7B and DeepSeek models achieved an AUC of 0.80, with CodeLlama 34B and WizardCoder 15B closely following. Each model demonstrates their effectiveness in differentiating between labels.

5.6 Computational Time Performance

The results in Table 2 show the mean computational time the CodeLlama models took to generate a response for pairing a GoVulnDB report with a vulnerable function. Report times were calculated based on the inference times for git-hunks within each report, with an average of five git-hunks per patch link per report. For instance, CodeLlama 7B averaged 0.31 seconds (s) per git-hunk, leading to a mean report time of 1.30s. The standard and detailed prompts have similar performance. Therefore, we only show the detailed prompt.

CoT prompts were, on average, 13 times (x) slower than the detailed prompt. Specifically, CoT prompts were 19x slower in zero-shot settings and 7x slower in few-shot settings. Overall, the few-shot approach was 2.4x slower than the zero-shot approach. Additionally, generation time doubles (2.1x) when moving from the smaller CodeLlama 7B model to the 13B model and a similar increase

Table 2: Comparing mean generation times (seconds) at report level with detailed vs. CoT prompts across CodeLlama models (7b/13b/34b) in zero-shot and few-shot settings. The Increase Factor CoT measures the slowdown with CoT prompts.

	Zero-Shot Time			Five-Shot Time		
Model	Detailed	CoT	Increase Factor CoT	Detailed	CoT	Increase Factor CoT
CodeLlama 7B	1.30s	38.80s	30x	4.57s	43.01s	9x
CodeLlama 13B	2.87s	46.85s	16x	10.13s	65.54s	6x
CodeLlama 34B	8.08s	88.05s	11x	28.78s	126.54s	4x

when moving from the 13B to the 34B model, regardless of the prompt or learning paradigm. These findings highlight the generation complexity of CoT prompts (discussed in Section 4.5), the inclusion of more text (i.e., few-shot), and larger models contribute to longer performance times.

6 Discussion

LLMs demonstrate a possible path forward for pairing security advisories with vulnerable functions, offering high-quality pairings with minimal manual effort, thereby enhancing industry vulnerability management efforts. Despite the potential, considerable scope exists for improvement, especially in increasing the overall recall of our analysis. Integrating program analysis with LLMs could improve recall. For example, a challenge for all models was identifying vulnerabilities addressed by introducing new functions, which are then invoked from within the existing vulnerable function. Such fixes are common in vulnerabilities requiring sanitization (e.g., CWE-20 Improper Input Validation, CWE-79 Cross-Site Scripting), where sanitization needs to be applied across multiple code locations. In such instances, additional context (e.g., surrounding code) would be required. Outside of code aspects, integrating commit messages and identifying additional links or missing patch links could enrich LLM effectiveness in the future. This could address cases where patch links are missing or vulnerable functions are located elsewhere. Another area for improvement is in the retrieval of examples. When the retrieval system finds similar examples between both *True* and *False* labels, the models have trouble discerning the correct results.

Threats to Validity: Our research is subject to both internal and external validity threats. Internally, we trust that the Google Go security team accurately identifies vulnerable functions in Go data. If these identifications are inaccurate, our results and evaluations could be inaccurate. The models we evaluated were also trained on public data, potentially including some of the Go code under investigation and GoVulnDB report data. However, our methodology, which pairs these security advisories with vulnerable functions from git-diff data, differs from the original training and testing approaches of these models, helping to mitigate the risk of data leakage. Externally, limiting our evaluation to the Go language could impact the generalizability of our findings. We expect similar performance across different languages, drawing on the generalization capabilities of LLMs [31,21,5,26]. Nevertheless, this assumption remains speculative without further empirical validation across a broader set of programming languages for pairing security advisories with vulnerable functions. This highlights a critical area for future research. However, it will take substantial effort first to accumulate high-quality data such as GoVulnDB in other languages.

7 Related Work

Vulnerability Datasets: Recent work [9,8] has found that commonly used vulnerability datasets have significant inaccuracy rates. These issues in such labels are due to automatic data collection and flawed semantic filters, underscoring the need for manual verification and comprehensive function analysis to assess vulnerability relevance accurately. For instance popular datasets such as CVEFixes [4], ReVeal [7], BigVul [15], CrossVul [27], and Devign [40] consider the changed functions within a security patch as vulnerable. Guo and Bettaieb found that vulnerability dataset quality (i.e., mislabeled data) can significantly degrade models that depend on noisy datasets [20]. Alongside our research, Wang et al. introduced ReposVul [35], a vulnerability dataset featuring CVE entries in 1,491 projects across four programming languages, detailed to the line level. ReposVul uses vulnerability untangling, using both LLMs and static analysis tools to separate vulnerability-related code changes from unrelated patch changes.

LLMs for Vulnerability Detection: The growing interest in LLMs for software vulnerability detection has significant insights, yet the field is still in its infancy. In several scenarios, Khare et al.[22] showed GPT-4's ability to outperform static analysis and other deep learning-based approaches in detecting vulnerabilities. Zhang et al.[39] assessed ChatGPT's performance on synthetic Java datasets and the CVEFixes dataset with various prompting strategies. However, Purba et al.[29] evaluated the efficacy of both open-source and proprietary LLMs in detecting SQL injection and buffer overflow vulnerabilities, noting a high incidence of false positives. Liu et al.[25] combined LLMs with binary taint analysis to uncover 37 new bugs in real-world firmware. The LLM4Vuln framework [32] focuses on identifying smart contract vulnerabilities by enhancing LLMs with additional knowledge, tool integration, and prompt engineering, uncovering nine zero-days. Other studies have explored using LLMs to generate fuzzing inputs [11] and integrating them with static analysis tools for new vulnerability discovery [24]. However, our interest lies in pairing security advisories with the corresponding vulnerable functions, differing from identifying new vulnerabilities.

8 Conclusion

In summary, this study highlights the challenge of automating the association between security advisories and vulnerable functions and demonstrates the potential of LLMs as a promising initial solution. Our findings suggest that while

CoT prompting enhances accuracy, it incurs computational costs. Alternatively, strategies using few-shot learning and concise prompts achieve comparable accuracy with reduced computational overhead. These advancements suggest a path to scale the automated matching of security advisories with vulnerable functions, which is key for enhancing vulnerability management processes.

Acknowledgments

This work is supported in part by NSF grants CNS-1946273 and CNS-2207008. Any findings and opinions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

1. Alahmadi, B.A., Axon, L., Martinovic, I.: 99% False Positives: A Qualitative Study of SOC Analysts' Perspectives on Security Alarms. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 2783–2800. USENIX Association (2022)
2. Alomar, N., Wijesekera, P., Qiu, E., Egelman, S.: "You've got your nice list of bugs, now what?" vulnerability discovery and management processes in the wild. In: Proceedings of the Sixteenth USENIX Conference on Usable Privacy and Security. SOUPS'20, USENIX Association, USA (2020)
3. Arora, A.: Understanding Logits, Sigmoid, Softmax, and Cross-Entropy Loss in Deep Learning — wandb.ai. https://wandb.ai/amanarora/Written-Reports/reports/Understanding-Logits-Sigmoid-Softmax-and-Cross-Entropy-Loss-in-Deep-Learning--Vmlldzo0NDMzNTU3 (2023)
4. Bhandari, G., Naseer, A., Moonen, L.: CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In: Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering. p. 30–39. PROMISE 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3475960.3475985
5. Bi, X., Chen, D., Chen, G., Chen, S., Dai, D., Deng, C., Ding, H., Dong, K., Du, Q., Fu, Z., et al.: Deepseek llm: Scaling open-source language models with longtermism. arXiv preprint arXiv:2401.02954 (2024)
6. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., et al.: Language models are few-shot learners. In: Proceedings of the 34th International Conference on Neural Information Processing Systems. NIPS '20, Curran Associates Inc. (2020)
7. Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: Are we there yet? IEEE Transactions on Software Engineering **48**(09), 3280–3296 (sep 2022). https://doi.org/10.1109/TSE.2021.3087402
8. Chen, Y., Ding, Z., Alowain, L., Chen, X., Wagner, D.: Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses. p. 654–668. RAID '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3607199.3607242
9. Croft, R., Babar, M.A., Kholoosi, M.M.: Data quality for software vulnerability datasets. In: Proceedings of the 45th International Conference on Software Engineering. p. 121–133. ICSE '23 (2023). https://doi.org/10.1109/ICSE48619.2023.00022

10. Dann, A., Plate, H., Hermann, B., Ponta, S., Bodden, E.: Identifying Challenges for OSS Vulnerability Scanners - A Study & Test Suite. IEEE Transactions on Software Engineering **48**(09), 3613–3625 (2022). https://doi.org/10.1109/TSE.2021.3101739
11. Deng, Y., Xia, C.S., Peng, H., Yang, C., Zhang, L.: Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 423–435. ISSTA 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3597926.3598067
12. Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.E., Lomeli, M., Hosseini, L., Jégou, H.: The faiss library. arXiv preprint arXiv:2401.08281 (2024)
13. Dunlap, T.: A python package for extracting commit features - patchparser (2022), https://github.com/tdunlap607/patchparser
14. EndorLabs: Endor Labs vs. SCA (Nov 2022), https://www.endorlabs.com/endor-labs-vs-sca
15. Fan, J., Li, Y., Wang, S., Nguyen, T.N.: A c/c++ code vulnerability dataset with code changes and cve summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories. p. 508–512. MSR '20, Association for Computing Machinery (2020). https://doi.org/10.1145/3379597.3387501
16. Google: Go Vulnerability Database, https://go.dev/doc/security/vuln/database
17. Google: Govulncheck, https://pkg.go.dev/golang.org/x/vuln/cmd/govulncheck
18. Google: Handling Go Vulnerability Reports, https://github.com/golang/vulndb/blob/master/doc/triage.md#add-a-new-report-label-needsreport
19. Google: Vulnreport, https://pkg.go.dev/golang.org/x/vulndb/cmd/vulnreport
20. Guo, Y., Bettaieb, S.: An Investigation of Quality Issues in Vulnerability Detection Datasets. In: 2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) (2023). https://doi.org/10.1109/EuroSPW59978.2023.00008
21. Jiang, A.Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D.S., Casas, D.d.l., Hanna, E.B., Bressand, F., et al.: Mixtral of experts. arXiv preprint arXiv:2401.04088 (2024)
22. Khare, A., Dutta, S., Li, Z., Solko-Breslin, A., Alur, R., Naik, M.: Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities. arXiv preprint arXiv:2311.16169 (2023)
23. LangChain: Custom example selector (2023), https://python.langchain.com/docs/modules/model_io/prompts/example_selectors/custom_example_selector
24. Li, H., Hao, Y., Zhai, Y., Qian, Z.: The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models. arXiv preprint arXiv:2308.00245 (2023)
25. Liu, P., Sun, C., Zheng, Y., Feng, X., Qin, C., Wang, Y., Li, Z., Sun, L.: Harnessing the power of llm to support binary taint analysis. arXiv preprint arXiv:2310.08275 (2023)
26. Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., Jiang, D.: WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv preprint arXiv:2306.08568 (2023)
27. Nikitopoulos, G., Dritsa, K., Louridas, P., Mitropoulos, D.: Crossvul: a cross-language vulnerability dataset with commit data. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1565–1569. ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3468264.3473122

28. Ponta, S.E., Plate, H., Sabetta, A.: Detection, assessment and mitigation of vulnerabilities in open source dependencies. Empirical Software Engineering **25**(5), 3175–3215 (2020). https://doi.org/10.1007/s10664-020-09830-x
29. Purba, M.D., Ghosh, A., Radford, B.J., Chu, B.: Software Vulnerability Detection using Large Language Models. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW). pp. 112–119 (2023). https://doi.org/10.1109/ISSREW60843.2023.00058
30. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al.: Language models are unsupervised multitask learners. OpenAI blog **1**(8), 9 (2019)
31. Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al.: Code Llama: Open Foundation Models for Code. arXiv preprint arXiv:2308.12950 (2023)
32. Sun, Y., Wu, D., Xue, Y., Liu, H., Ma, W., Zhang, L., Shi, M., Liu, Y.: Llm4vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning. arXiv preprint arXiv:2401.16185 (2024)
33. TreeSitter: An incremental parsing system for programming tools - tree-sitter (2023), https://tree-sitter.github.io/tree-sitter/
34. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L.u., Polosukhin, I.: Attention is All you Need. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc. (2017)
35. Wang, X., Hu, R., Gao, C., Wen, X.C., Chen, Y., Liao, Q.: Reposvul: A Repository-Level High-Quality Vulnerability Dataset. arXiv preprint arXiv:2401.13169 (2024)
36. Wang, Y., Le, H., Gotmare, A.D., Bui, N.D., Li, J., Hoi, S.C.: Codet5+: Open code large language models for code understanding and generation. arXiv preprint arXiv:2305.07922 (2023)
37. Wei, J., Bosma, M., Zhao, V.Y., Guu, K., Yu, A.W., Lester, B., Du, N., Dai, A.M., Le, Q.V.: Finetuned Language Models Are Zero-Shot Learners. arXiv preprint arXiv:2109.01652 (2021)
38. Wei, J., Wang, X., Schuurmans, D., Bosma, M., ichter, b., Xia, F., Chi, E., Le, Q.V., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models. In: Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., Oh, A. (eds.) Advances in Neural Information Processing Systems. vol. 35, pp. 24824–24837. Curran Associates, Inc. (2022)
39. Zhang, C., Liu, H., Zeng, J., Yang, K., Li, Y., Li, H.: Prompt-Enhanced Software Vulnerability Detection Using ChatGPT. arXiv preprint arXiv:2308.12697 (2023)
40. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 32. Curran Associates, Inc. (2019)