Large Language Models for In-File Vulnerability Localization Can Be "Lost in the End"

FRANCESCO SOVRANO, ETH Zurich, Switzerland and University of Zurich, Switzerland
ADAM BAUER, University of Zurich, Switzerland
ALBERTO BACCHELLI, University of Zurich, Switzerland

Traditionally, software vulnerability detection research has focused on individual small functions due to earlier language processing technologies' limitations in handling larger inputs. However, this function-level approach may miss bugs that span multiple functions and code blocks. Recent advancements in artificial intelligence have enabled processing of larger inputs, leading everyday software developers to increasingly rely on chat-based large language models (LLMs) like GPT-3.5 and GPT-4 to detect vulnerabilities across entire files, not just within functions. This new development practice requires researchers to urgently investigate whether commonly used LLMs can effectively analyze large file-sized inputs, in order to provide timely insights for software developers and engineers about the pros and cons of this emerging technological trend. Hence, the goal of this paper is to evaluate the effectiveness of several state-of-the-art chat-based LLMs, including the GPT models, in detecting in-file vulnerabilities. We conducted a costly investigation into how the performance of LLMs varies based on *vulnerability type*, *input size*, and *vulnerability location* within the file. To give enough statistical power ($\beta \geq .8$) to our study, we could only focus on the three most common (as well as dangerous) vulnerabilities: XSS, SQL injection, and path traversal. Our findings indicate that the effectiveness of LLMs in detecting these vulnerabilities is strongly influenced by both the location of the vulnerability and the overall size of the input. Specifically, regardless of the vulnerability type, LLMs tend to significantly ($p < .05$) underperform when detecting vulnerabilities located toward the end of larger files—a pattern we call the 'lost-in-the-end' effect. Finally, to further support software developers and practitioners, we also explored the optimal input size for these LLMs and presented a simple strategy for identifying it, which can be applied to other models and vulnerability types. Eventually, we show how adjusting the input size can lead to significant improvements in LLM-based vulnerability detection, with an average recall increase of over 37% across all models.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → **Neural networks**; • **Security and privacy** → *Software security engineering*.

Additional Key Words and Phrases: Large Language Models, In-File Vulnerability Detection, XSS, SQL Injection, Path Traversal, 'Lost-in-the-End' Issue, Code Context

ACM Reference Format:
Francesco Sovrano, Adam Bauer, and Alberto Bacchelli. 2025. Large Language Models for In-File Vulnerability Localization Can Be "Lost in the End". *Proc. ACM Softw. Eng.* 2, FSE, Article FSE041 (July 2025), 23 pages. https://doi.org/10.1145/3715758

1 Introduction

In the rapidly evolving field of software development, the integration of large language models (LLMs) has shifted from a novel concept to a mainstream practice. As generative artificial intelligence

Authors' Contact Information: Francesco Sovrano, Collegium Helveticum, ETH Zurich, Zurich, Switzerland and University of Zurich, Department of Informatics, Zurich, Switzerland, sovrano@collegium.ethz.ch; Adam Bauer, University of Zurich, Zurich, Switzerland, adam.bauer@uzh.ch; Alberto Bacchelli, University of Zurich, Department of Informatics, Zurich, Switzerland, bacchelli@ifi.uzh.ch.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2994-970X/2025/7-ARTFSE041
https://doi.org/10.1145/3715758

(AI) technologies continue to advance, tools like ChatGPT, Copilot, and similar models are now embedded as first-class citizens into development environments, assisting with tasks such as code generation, bug fixing, and security analysis. According to Sonatype's 2023 report [40], 97% of developers and security professionals now rely on LLMs in their workflows, marking a significant shift in how software vulnerabilities are detected and patched.

Indeed, the integration of LLMs addresses a critical challenge identified in the 2022 GitLab Survey [17], which noted that developers often fail to detect security-relevant bugs early and do not prioritize bug fixing adequately. This typically leads to frequent vulnerabilities that are challenging for humans to detect. Instead, LLMs have been found to be quite effective in supporting vulnerability detection [5, 16, 43, 51]. This efficacy is primarily attributed to their capability to interpret code semantics "like a human" [30], an advantage not typically possessed by fuzzers and classical static analyzers.

Historically, the detection of software vulnerabilities has primarily focused on analyzing individual functions due to limitations in earlier natural language processing and machine learning models [8, 9, 23, 31, 31, 48, 50, 51]. However, this narrow scope can overlook vulnerabilities that span across multiple functions or different parts of a file (as in the example shown in Figure 1), leading to an incomplete or missed detection [44].

Recent advancements in LLMs have significantly expanded their capacity to process large portions of code, now reaching lengths of millions of characters [7, 12, 34, 49]. These models can now analyze entire files or even entire repositories. As a result, software developers are increasingly relying on chat-based LLMs like GPT-3.5 and GPT-4 to detect vulnerabilities across whole files. Given this growing reliance, it is crucial to assess whether these commonly used state-of-the-art chat-based LLMs can effectively identify vulnerabilities in large, file-sized inputs. Such an evaluation will provide software developers and engineers with timely insights into the strengths and limitations of this emerging technology, helping them make informed decisions about its use.

In-file vulnerability detection is crucial because many vulnerabilities involve multiple functions. Of the top-10 most dangerous security vulnerabilities in 2024 [29], at least six frequently span functions: 'Out-of-bounds Write' (#2) and 'Out-of-bounds Read' (#6) involve buffer allocation and bounds checking in different file areas, 'Cross-site Scripting', 'SQL Injection', and 'Path Traversal' (#1, #3, #5) span input sanitization handling across modules, 'Use After Free' (#8) involves memory allocation, freeing, and reuse in separate areas, and 'Missing Authorization' (#9) spans missing authorization checks across access points.

This leads to important questions: Can LLMs safely detect vulnerabilities within files that are up to the size of their context windows, or do intrinsic limitations exist that necessitate the use of smaller inputs? If smaller inputs are required, how small must they be? This study seeks to answer these questions, offering practitioners a clearer understanding of the capabilities and limitations of LLMs in detecting vulnerabilities within large code-bases.

We thus conducted several empirical studies involving open-source LLMs like Mixtral 8x7b, Mixtral 8x22b, and Llama 3 70B, as well as well-known commercial models such as GPT-3.5, GPT-4, and GPT-4o. These chat-based models were employed off-the-shelf (i.e., without customization, as a typical software developer might use them) and tested across a range of scenarios involving real-world security vulnerabilities from the public Common Vulnerabilities and Exposures (CVE) catalog [28], which lists software vulnerabilities categorized under the Common Weakness Enumeration (CWE), a standardized system for identifying software weaknesses.

To assess the vulnerability detection capabilities of these LLMs in this realistic setting, we used a balanced dataset, which we compiled from the CVE catalog by following established best practices in the vulnerability localization literature [8, 9, 31, 43, 45, 48]. Specifically, the dataset we assembled

comprises over 750 files containing vulnerabilities, along with over 750 "non-vulnerable" files obtained by applying the patches for these vulnerabilities as provided in the CVE catalog.

We ensured that the dataset was constructed such that each file represented a unique vulnerability instance. Before expanding the study to multi-file systems or entire repositories, we first focused on single-file vulnerability detection.

Additionally, to achieve sufficient statistical power ($\beta \geq .8$) in our study, we considered only three common CWE types: cross-site scripting (XSS), SQL injection, and path traversal. These were the only CWE types for which we identified at least 100 distinct vulnerability instances, a number determined by *a priori* power analysis [13].

We then used this dataset to measure the effectiveness of each LLM at detecting vulnerabilities using precision, recall, accuracy, and F1 score metrics. Afterwards, we examined how file size and the position of the vulnerability within the file impact the effectiveness of the LLMs. This part of the study involved two key experiments. The first experiment assessed the correlation between input size and the models' detection accuracy across the over 750 real-world vulnerabilities we collected from the CVE catalog. The second experiment, referred to as the 'code-in-the-haystack' experiment, specifically investigated how relocating the same vulnerable lines of code within files of varying sizes affects the models' detection capabilities. Through these experiments, we identified and quantified a 'lost-in-the-end' issue, where well-known LLMs such as ChatGPT tend to miss vulnerabilities located at the end of long files.

This finding contrasts with the common trend identified in the literature, where LLMs are often observed to suffer from a 'lost-in-the-middle' issue (i.e., they process information well at the beginning and end, but not in the middle), rather than a 'lost-in-the-end' problem [1, 22, 24, 26, 47]. While we did not analyze additional types of vulnerabilities, the trends observed suggest that the 'lost-in-the-end' issue may also affect other vulnerability types. Regardless, the three vulnerabilities we studied are among the most prevalent and dangerous in web security, making our findings highly relevant to the community.

Building on these findings, we empirically investigated how small an input code needs to be to maximize the probability of detecting a vulnerability with one of the considered LLMs, thereby mitigating the 'lost-in-the-end' issue. This involved using simple file chunking strategies to adjust the dataset file sizes and analyzing how these changes influenced the models' ability to correctly identify vulnerabilities. By observing the resulting detection scores, we identified more effective input configurations for each LLM and vulnerability type.

This knowledge should help practitioners better understand how to effectively use LLMs, such as ChatGPT and Llama, in practice for vulnerability detection.

2 Background and Related Work

This section compares our study with other research on language models for vulnerability detection and explores the relationship to prior natural language processing research, focusing on the 'lost-in-the-middle' issue and how input size affects LLMs performance.

2.1 Vulnerability Detection via LLMs

There has been extensive research on the application of language models for vulnerability detection in software development and maintenance, primarily involving small specialized models operating at a function-level granularity [8, 9, 23, 31, 48, 50, 51]. The recent advent of ChatGPT has shifted some focus to the use of LLMs, but still at a function-level granularity [38, 41, 42, 51].

The majority of the aforementioned studies used vulnerability datasets extracted from the CVE catalog, a practice we also adopt in our research. The CVE MITRE catalog is a public database managed by the MITRE Corporation that lists and describes software and firmware vulnerabilities,

assigning them unique identifiers for standardized handling. The catalog classifies vulnerabilities using the Common Weakness Enumeration, which categorizes software and hardware weaknesses with security implications. Each CWE entry details a specific type of vulnerability, helping organizations to better identify, understand, and mitigate potential threats. An example of CVE entry of type CWE-22, wherein the fix spans multiple functions. is shown in Figure 1.

```
@@ -52,9 +52,13 @@ public function createElements(array $formData)
54  54            $callbackValidator = new Zend_Validate_Callback(function ($value) {
55      -             if (openssl_pkey_get_private($value) === false) {
    55  +             if (
    56  +                 substr(ltrim($value), 0, 7) === 'file://'
    57  +                 || openssl_pkey_get_private($value) === false
    58  +             ) {
56  59                    return false;
@@ -126,20 +130,19 @@ public static function beforeAdd(ResourceConfigForm $form)
126 130       $configDir = Icinga::app()->getConfigDir();
127 131       $user = $form->getElement('user')->getValue();
128 132
129     -       $filePath = $configDir . '/ssh/' . $user;
130     -
    133 +       $filePath = join(DIRECTORY_SEPARATOR, [$configDir, 'ssh', sha1($user)]);
```

Fig. 1. GitHub commit for CVE-2022-24715, showing a security patch to prevent a path traversal.

Studies such as the one by Ullah et al. [43] have shown significant non-robustness in advanced models like PaLM2 and GPT-4. Changes in function or variable names, or the addition of library functions, can cause these models to yield incorrect answers in 26% and 17% of cases, respectively. This highlights the need for further advancements before LLMs can serve as reliable security assistants. Although Ullah et al.'s work [43] studies code augmentation, unlike ours it does not explore the relationship between bug positions and the LLMs's ability to identify vulnerabilities.

Instead, Chen et al. [5] found that ChatGPT's performance varies across different vulnerabilities and bugs, and multi-round conversations often worsens detection rates. However, they do not investigate the underlying reasons for this, which we attribute to the 'lost-in-the-end' problem, where more context leads to information being lost.

A more recent work by Zhou et al. [51] has shown that with specific prompts (containing examples of the bug), ChatGPT can outperform smaller specialized models like CodeBERT [14] and CodeT5 [46] in detecting vulnerabilities. However, their analysis is limited to function-level detection and simple yes-or-no questions without explicit localization of the main line of code responsible for the vulnerability. Our study extends to in-file detection, using post-fix code as negatives, making the detection task more challenging and realistic.

Zhou et al. [51] try several prompts to understand their impact in vulnerability detection, also suggesting that vulnerabilities belonging to less common CWE types (e.g., CWE-22; path traversal) are also less likely to be identified by LLMs. Indeed, according to Zhou et al. [50], vulnerability data exhibit a long-tailed distribution in terms of CWE types: a small number of CWE types have a substantial number of samples (e.g., CWE-79; XSS), while numerous CWE types have very few samples. However, with our experiments we empirically show that with LLMs, the most frequent weaknesses (e.g., CWE-79) are actually those less likely to be identified by the model compared to less frequent ones (e.g., CWE-22).

2.2 'Lost-in-the-Middle' Issue and Input Size Impact on LLMs

Although many modern LLMs, including ChatGPT, can process extensive inputs, they often struggle to effectively manage information within lengthy contexts. This challenge has been highlighted in many other studies sometimes referring to a 'lost-in-the-middle' issue [24]. 'Lost-in-the-middle' refers to a phenomenon in large language models where information presented in the middle of a long input sequence is more likely to be overlooked or forgotten compared to information at the beginning or end.

These problems seem to happen because these language models do not know well how to identify relevant details amid fairly large contexts, and this has an impact on their ability to consistently apply contextual knowledge over extended sequences of information, reasoning over code and locating vulnerabilities.

To identify and analyze the 'lost-in-the-middle' issue, researchers have recently started to use the 'needle-in-the-haystack' experiment. In this approach, a specific piece of information, known as the 'needle', is hidden within a large block of irrelevant or filler text, referred to as the 'haystack'. The model is then tasked with finding and retrieving this specific information. This method was recently popularized online by specialized blogs [11] and it is now used to assess the memory capabilities of LLMs. Since its introduction, several gray-literature papers (i.e., non-peer-reviewed arXiv preprints) have adopted the needle-in-the-haystack method to evaluate LLMs' ability to recall information and effectively use input memory. Notable examples include works by An et al. [1], Li et al. [22], Liu et al. [24], Machlab and Battle [26], Xu et al. [47].

Typically, the standard needle-in-the-haystack experiment only requires a LLM to identify a specific piece of information within the input and does not involve reasoning over the entire content, as required in vulnerability detection. Therefore, we performed an adaptation (which we named 'code-in-the-haystack') of this approach specifically for vulnerability localization, requiring reasoning over an entire code context. This context is not just random text or code, but related code taken from the same file and repository.

Our results indicate that the LLMs we considered for vulnerability localization do not really suffer from the 'lost-in-the-middle' issue but rather from a 'lost-in-the-end' issue. This is aligned with the findings of Levy et al. [21] which show that large input sizes can hinder the reasoning capabilities of LLMs. However, Levy et al. [21] focused on general reasoning tasks unrelated to coding, whereas our study builds on this by exploring the impact of input size on the ability of LLMs to detect specific kinds of vulnerabilities within large code contexts.

3 Research Questions

Recent advancements in artificial intelligence have made it possible for LLMs to process entire files or even repositories, rather than just small code segments. As the popularity of chat-based LLMs continues to grow and the use of these LLMs for in-file vulnerability detection becomes more widespread among software developers, it is crucial to assess their effectiveness across various vulnerability types to inform practitioners of the pros and cons of these technological trends. This urgency led us to our initial research question:

RQ1: To what extent can popular chat-based LLMs detect vulnerabilities within entire files?

To address this question, we first obtained a dataset containing an equal distribution of vulnerable and patched (i.e., without the vulnerability) files for different CWE types. As detailed in Section 4, the dataset was extracted from the CVE catalog (which includes known vulnerabilities and

their fixes/patches; cf. Section 2.1), by selecting all vulnerabilities wherein the patch targets a single file of code. Following the methodology described in Section 5, we then instructed popular state-of-the-art LLMs such as GPT-4 to find and return any line within the dataset files that may lead to a vulnerability of a given CWE type. This approach allowed us to compute metrics such as precision, recall, accuracy, and F1 score for each LLM and CWE type.

Once we determine the effectiveness of these models in detecting vulnerabilities within files, we can investigate the reasons behind their errors to inform practitioners about best practices. To do this, we decided to examine whether the position of the vulnerability or the file size impacts the LLMs' performance, specifically assessing whether they experience any 'lost-in-the-middle' issues (cf. Section 2.2). This leads to our next question:

RQ2: To what extent input size and vulnerability position affect LLMs' ability to detect vulnerabilities?

To answer it, we conducted two experiments. The first experiment (cf. Section 6.1) used the dataset from RQ1 to examine the correlation between input size, the position of the vulnerability, and the correctness of the LLM's output, focusing only on the files containing a vulnerability.

However, this first experiment presented a problem: the distribution of actual file sizes and vulnerability locations follows a tailed distribution, and this could distort the results. Therefore, we conducted a second experiment (whose methodology is detailed in Section 6.2), which complements the previous one by uniforming the distribution of vulnerability locations and file sizes. Consequently, this second experiment was designed to more effectively determine whether a given LLM suffers from a 'lost-in-the-middle' or 'lost-in-the-end' problem.

Finally, building on the insights from RQ2, we aimed to establish best practices for practitioners when using mainstream LLMs in cases where they suffer from the 'lost-in-the-middle' issue or similar problems. If these LLMs have intrinsic limitations that hinder the safe detection of vulnerabilities in files approaching the size of their context windows, we seek to provide practitioners with guidance on the maximum input size that still ensures reliable detection capabilities. This led to the formulation of our third research question:

RQ3: If smaller inputs are required, how small must they be?

We hypothesize that the optimal input size could be determined by analyzing how variations in input size affect the LLM's ability to accurately identify the location of a vulnerability. To test this, we divided the vulnerable files into chunks of varying sizes, following the methodology presented in Section 7 and adapting the experimental method from RQ1 to assess file chunks rather than entire files. After determining the optimal input size for an LLM and CWE type, practitioners can chunk files accordingly, improving vulnerability detection likelihood within their code.

4 Dataset Construction

To address RQ1, we need a dataset with an equal distribution of vulnerable and non-vulnerable files, categorized by CWE type, where each vulnerability fix affects only a single file. This ensures that the LLM has sufficient context to understand the vulnerability. As no such dataset was available in the related literature (see Section 2), we created one ourselves by extracting data from the CVE catalog, as has been done in related studies [8, 9, 31, 43, 45, 48]. Indeed, the CVE catalog contains thousands of entries, each linking to a specific vulnerability and its corresponding patch (i.e., the vulnerability fix).

Table 1. Dataset Statistics: file counts, median file and function sizes (in chars), global statistics for 2022 (frequency and danger of CWE in the National Vulnerabilities DB).

CWE-ID	No. Files	Func. per File	Func. Size	File Size	File Size Quartiles	Danger Score	NVD Count
CWE-22	105	8	1,073	7,534	(3,715-14,839)	14.11	1,010
CWE-89	146	8	948	6,255	(3,213-19,007)	22.11	1,263
CWE-79	543	9	1,005	8,654	(3,819-17,331)	45.97	4,740

Methodology. Following an established procedure in vulnerability localization research [8, 9, 31, 43, 45, 48], we labeled the file before the patch as the “vulnerable” file and the file after the patch as the “non-vulnerable” file. This ensures that the pre-fix code contains a vulnerability that the LLM must detect, while the post-fix code should be free of the same vulnerability. Although Rice’s theorem [39] suggests it is impossible to determine whether a piece of code is entirely free of vulnerabilities, we use these “non-vulnerable” files only for answering RQ1, which benchmarks LLMs. This does not affect our core findings and contributions which stem from RQ2 and RQ3.

We initiated the data extraction process by scraping all CVE entries from March 2018 to March 2024, covering a period of six years. These entries were filtered to include only those referring to single GitHub commits affecting a single source code file, excluding documentation files. For instance, if a commit targeted one code file and two documentation files, it was included, whereas commits containing multiple code files were not. Eventually, we only considered files written in: PHP, TypeScript, JavaScript, HTML, Java, Go, Python, Ruby, and C.

Consequently, we excluded file extensions associated with documentation or data, such as txt, svg, md, xml, and json. We also excluded file extensions associated with C++, including header files. Since RQ1 is on in-file vulnerability localization, this focus on single-file changes ensures that the quality of the findings is not compromised by the fact we did not consider more files per vulnerability, such as entire repositories or C++ software (comprising both .cpp and .h files).

Since this dataset will be used for several statistical tests (e.g., logistic regressions), we included only CWE types with at least 100 CVE entries to ensure the tests have sufficient power ($\beta \geq .8$), as indicated by an *a priori* power analysis [13]. This analysis assumes a uniform distribution of bug positions, a large effect size of bug position or file size on vulnerability detection, and an $\alpha < .05$. This choice eventually ensures that our results are representative and based on a sufficient number of vulnerabilities, as all metrics in our experiments are computed separately for each CWE type.

Results. Using the methodology provided above, we retrieved 794 vulnerable files across three CWE types. For additional statistics, see Table 1. These types, ordered by prevalence of single-file patches in the CVE catalog, are:

- CWE-79: XSS, the second most dangerous[1] CWE type in 2022 and the category with the highest number of single-file patches.
- CWE-89: SQL Injection, the third most dangerous CWE type.
- CWE-22: Path Traversal, ranked eighth, but the category with the third highest number of single-file patches; identified as a rarer or ’tail’ weakness by [50].

All these CWE types fall under the group of “error prone processing of data originating from untrusted sources that often results in an initial entry point for an attacker to compromise an IT system”. Eventually, our final dataset comprises 794 vulnerable files and the same 794 files with the vulnerability patched (as provided in the CVE catalog), for a total of 1,588 unique files.

[1]The level of danger presented by a particular CWE is determined by multiplying the severity score by the frequency score.

Table 2. Selected LLMs. This table provides several statistics for each model.

Model	Know. Cut-off	Size	Open Source	Max. Tokens	Max. Characters
mixtral-8x7b	2023-12	45B	Yes	32k	~120k
mixtral-8x22b	2023-12	141B	Yes	64k	~250k
llama-3-70b	2023-12	70B	Yes	8k	~30k
gpt-3.5-turbo	2021-09	-	No	16k	~60k
gpt-4-turbo	2023-12	-	No	128k	~500k
gpt-4o	2023-10	-	No	128k	~500k

5 RQ1: LLMs vs. In-File Vulnerability Localization

The first research question aims to benchmark popular chat-based LLMs on in-file vulnerability localization tasks.

Methodology. For our analysis, we selected three cutting-edge open-source models (Mixtral 8x7b, Mixtral 8x22b, and Llama 3 70b) and three well-known commercial LLMs from OpenAI: ChatGPT 3.5 (version 0125), ChatGPT 4 (version 2024-04-09), and ChatGPT 4o (version 2024-05-13). The Mistral models have gained significant popularity as the first state-of-the-art open-source LLMs licensed under Apache 2.0. Llama 3, on the other hand, is the flagship open-source LLM from Meta, widely recognized in the AI community and arguably one of the most effective LLMs for vulnerability detection in the open-source landscape [3, 25], as well as the most mainstream. Finally, the ChatGPT models, with over 180 million users, are the most widely used commercial LLMs on the market and are the first choice for many developers, as they consistently outperform other general-purpose chat-based LLMs in many (if not most) vulnerability detection benchmarks [3, 25, 41, 43]. For more details about the selected LLMs, including their size and context window, please refer to Table 2.

Throughout our experiments, we used these models in their default configurations without any fine-tuning. This approach was chosen to assess the general applicability and effectiveness of these models as they are, i.e., without customization, similar to how a typical software developer might use them. Indeed, our goal is to investigate whether commonly used LLMs can effectively analyze large file-sized inputs, to provide timely insights for software developers and engineers. Additionally, by avoiding model-specific fine-tuning, we can also understand the baseline capabilities of these LLMs in vulnerability detection without the confounding effects of specialized training, which may not be fully replicable or practical in many real-world scenarios.

The only default configuration hyperparameter we changed for all the LLMs was their *temperature*. The temperature of an LLM is a hyperparameter that controls the randomness of its output. It can have any positive real value, with lower values making the output more focused and deterministic, and higher values making it more diverse and creative. We set the temperature to 0, the lowest possible value. Setting an LLM's temperature to 0 for vulnerability detection ensures consistent, deterministic, and repeatable responses, which is critical for accurately identifying and analyzing security flaws with little unwanted variability in the results.

With each LLM, we used the same prompt, engineered to provide an example of the expected output. This prompting strategy is called *in-context learning* [27, 52]. Differently from [51], we always used the same example across all tested vulnerability types, specifically a CWE-79 bug involving a 'user_input' JavaScript variable being concatenated into HTML content without proper sanitation.

This is the prompt we used:

Analyze the file content below and tell me if there’s any line that may contain a bug of type CWE-{bug_type_id} ({bug_type_label}). Your output must adhere to the following structure.

Expected Output Structure:
SE: very Short Explanation of why the line may contain a bug of given type (e.g., The ’user_input’ is directly concatenated into HTML content without sanitation).
BL: the Bugged Line, if any is found, else none (e.g., ‘response = "<html><body><h1>Welcome, " + user_input + "!</h1></body></html>"‘).
BUG FOUND: YES if a bug is found, else NO.

Example output:
SE: The ’user_input’ is directly concatenated into HTML content without sanitation.
BL: ‘response = "<html><body><h1>Welcome, " + user_input + "!</h1></body></html>"‘
BUG FOUND: YES

File Content:
{file_content}

The required output structure comprises a yes-or-no answer regarding whether the file contains a vulnerability, followed by a brief explanation detailing the rationale behind the potential bug presence in the specified line. For instance, the explanation might note that the ’user_input’ variable is directly concatenated into HTML content without proper sanitation. Additionally, the LLM is instructed to also copy and paste the identified problematic line in the output.

The designed prompt explicitly requires the models to analyze the contents of the file and determine if any line might contain a particular bug of the CWE type initially associated (in the CVE catalog) with the file. For example, for the “SshResourceForm.php” file of CVE-2022-24715 shown in Figure 1, either before or after the patch, the LLMs are asked only if any CWE-22 type of vulnerability is present. They are not asked about CWE-89 or CWE-79.

Following prior work, we measure performance using top-1 accuracy, a standard metric for multi-line vulnerabilities [2,3,4]. A true positive occurs when the model correctly answers "yes" and identifies at least one vulnerable line. "yes" answers without matching lines are false positives, while incorrect "no" answers are false negatives.

To evaluate whether an LLM correctly identified a vulnerability, we followed prior work and measured performance using top-1 accuracy—a standard metric for multi-line vulnerabilities [15, 32, 36]—by examining the yes-or-no answer provided by the model. If the model incorrectly answers ‘yes’, we classify the output as a false positive. Conversely, if the model incorrectly answers ‘no’, we classify the output as a false negative. A correct ‘no’ answer is marked as a true negative. For all other cases, differently from [51], we compare the line content reported by the model to the patch, ensuring normalization of newlines, white-spaces, and tabs. If the model identifies at least one line that exists in the patch diff (i.e., a line that was changed as part of the bug fix), it means the model correctly detected the buggy line. This is classified as a true positive. Else, if the model fails to detect any line that appears in the patch diff, the output is classified as a false negative.

Results. Table 3 summarizes the accuracy, F1 scores, precision, and recall across all the considered CWE and LLMs. These scores underline the following finding:

Finding 1: Off-the-shelf LLMs demonstrate low and uneven accuracy across vulnerability types.

The data shows that commercial models generally exhibit superior performance metrics in comparison to their open-source counterparts. Specifically, the models ChatGPT-4 (turbo) and ChatGPT-4o consistently achieve the highest scores across all metrics for CWE-22 and CWE-79.

Table 3. Performance summary by CWE and model. For each CWE, the best scores column-wise are in bold.

CWE-ID	Model	F1	Accuracy	Precision	Recall	Files
CWE-22	mixtral-8x7b	.305	.195	.268	.352	210
	mixtral-8x22b	.402	.290	.347	.476	210
	llama-3-70b	.360	.235	.309	.430	200
	gpt-3.5-turbo	.339	.212	.292	.404	208
	gpt-4-turbo	**.470**	**.324**	**.387**	**.600**	210
	gpt-4o	.448	.295	.368	.571	210
CWE-89	mixtral-8x7b	.281	.181	.250	.322	287
	mixtral-8x22b	.362	.240	.312	.432	292
	llama-3-70b	.400	**.276**	**.343**	.481	257
	gpt-3.5-turbo	.386	.255	.328	.469	286
	gpt-4-turbo	.383	.250	.325	.466	292
	gpt-4o	**.410**	.260	.341	**.514**	292
CWE-79	mixtral-8x7b	.173	.126	.164	.183	1084
	mixtral-8x22b	.220	.161	.205	.236	1084
	llama-3-70b	.226	.200	.219	.233	985
	gpt-3.5-turbo	.197	.217	.202	.192	1061
	gpt-4-turbo	.246	**.253**	**.248**	.243	1086
	gpt-4o	**.275**	.184	.247	**.309**	1086

For CWE-89, however, the open-source model Llama 3 surfaces as a strong competitor, in terms of accuracy and precision, surpassing the commercial models except ChatGPT-4o in recall and F1.

Focusing on the open-source models, Llama 3 outperforms the Mixtral series in all evaluated categories and across each CWE type, asserting its efficacy among non-commercial options. While the Mixtral 8x22b demonstrates a noticeable improvement over Mixtral 8x7b, particularly in F1 scores and precision, it still falls short of the performance standards set by Llama 3.

Despite variations in performance across different models and CWE categories, the results suggest that none of the models achieve particularly high metrics, indicating a room for significant improvement. For example, the highest accuracy recorded across all models and categories is only .324, a value for the commercial model ChatGPT-4 (turbo) in CWE-22, which is relatively modest in fields where high accuracy and reliability are crucial. Similarly, the best F1 score observed is .470, also by ChatGPT-4 (turbo) for CWE-22, suggesting that even the best model's ability to balance precision and recall is limited. The open-source models, particularly Llama 3, show promising results but still do not reach a level of performance that could be considered highly effective, with their best accuracy peaking at .276 in CWE-89.

Since LLMs are known to perform differently across programming languages [4, 20], we also provide an analysis based on the programming language distribution encountered in our dataset. The dataset comprises PHP (1020 instances), JavaScript (214 instances), Python (112 instances), and Java (96 instances) files, with smaller contributions from Go, Ruby, and others. A statistical analysis of LLM performance across the top four languages using the Kruskal–Wallis test revealed no significant differences in correctness. In particular, the following results were obtained: Mixtral-8x7b yielded $H = 1.661$ with $p = .436$; mixtral-8x22b ($H = 1.258; p = .533$); llama-3-70b ($H = 3.122; p = .210$); gpt-3.5-turbo ($H = 1.140; p = .566$); gpt-4-turbo ($H = .936; p = .626$).

6 RQ2: The Impact of Vulnerability Position and File Size

Our study examines how the location of vulnerabilities within a file and the file size influence the performance of state-of-the-art LLMs in detecting the three CWE types identified in Section 5. This section details the two experiments conducted to address RQ2 as outlined in Section 3.

6.1 File Size and Position of Bug vs. Probability of Detection on Real-World Files

The first experiment examines the correlation between input size, the position of the vulnerability, and the ability of the LLM to detect the vulnerability within the 794 vulnerable files in our dataset.

Methodology. Similarly to the previous experiments, also in this case we fed hundreds of vulnerable files to the selected LLMs, asking them to detect and locate the vulnerabilities based on their CWE type. We then studied how well those LLMs did the job, performing an analysis to see whether there is a correlation between their performance and the location of the vulnerabilities as well as the size of the file.

Since the LLM's output can only be correct or incorrect, for the correlation analysis we employed logistic regressions, segregating the results by CWE type. Specifically, we used *simple logistic regression* models [35] to analyze vulnerability detection as the dependent variable, with either bug position or file size as independent predictors. We chose logistic regression over ANOVA or the Mann-Whitney U test due to the dichotomous nature of the dependent variable and non-normal distribution of predictors (Figure 2, left).

The test statistic for assessing the significance of each logistic regression coefficient is a z-test, as is typically the case [10]. In our replication package [2], we follow APA guidelines to report the regression coefficients, 95% confidence intervals, effect sizes (i.e., odds ratios), and p-values for each model term (intercept and predictors). Additionally, we also include the results of a *multiple logistic regression* (i.e., combining both predictors), which remain consistent with those obtained from *simple logistic regression*.

We measured file size by the total number of characters and determined the position of the vulnerability by counting the characters preceding the vulnerability (as identified through the patch). Although LLMs process inputs by breaking them into tokens, each LLM uses a distinct tokenization strategy. Therefore, to maintain comparability across distinct LLMs, we opted to measure using characters rather than tokens. Indeed, measuring effects by character count implies effects in token and line counts, as both are composed of characters.

Results. The results of the logistic regressions, presented in Figure 2, reveal that vulnerabilities positioned deeper or within larger files are significantly less likely to be identified by LLMs. These trends persist across all examined CWE types, with p-values significantly below .05 in all instances.

A notable pattern emerges when combining these logistic regressions with the CWE vulnerability frequency in the CVE catalog, as shown in Table 1. The data indicate that the most frequently occurring CWE type in our dataset (CWE-79) experiences the deepest decline in vulnerability localization probability. Additionally, in the commercial models (the GPTs), we observe that the negative impact of file size on detection probability is proportional to the frequency of the CWE type. Specifically, the line representing CWE-22 (the least frequent type in our dataset) is generally above that of CWE-89, which is above CWE-79.

These trends mirror the observed accuracy scores (see Table 3). In other words, the trends observed by [50] and mentioned by [51] where frequent vulnerabilities are more easily detected/addressed do not hold true in our specific experiments and are actually inverted.

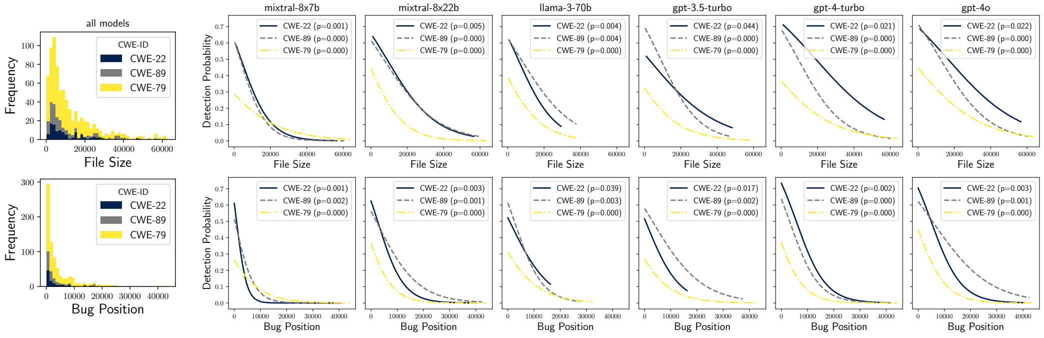

Fig. 2. Real-World Files: Logistic regressions across LLMs and CWE types; data distributions on the left.

6.2 Code-in-the-Haystack Experiment

Given that the probability of randomly identifying a vulnerable line decreases with increasing bug location and file size and that (as shown in Figure 2) the bug location and file size follow a tailed distribution, we conducted a follow-up experiment. This second experiment aims to determine whether the previously shown results are due to the inability of the LLMs to effectively process contextual information or to the tailed distribution of vulnerability features.

Methodology. In the second experiment, called the 'code-in-the-haystack' experiment, we focused on a limited number of different vulnerability instances, specifically five per CWE type. For each instance, we relocated the vulnerable lines of code within the same file, varying the file sizes to create over 500 different file combinations per CWE type, to further verify whether the LLMs' detection capabilities correlate with file size and bug position. Similarly to the other RQ2 experiment, we then used the same *simple logistic regressions* to analyze vulnerability detection. This experiment was named after the 'needle-in-the-haystack' experiments discussed in Section 2.

In other words, our code-in-the-haystack experiment involves selecting a set of vulnerabilities for each considered CWE type and systematically altering the position of the vulnerability's core line within the file so that the distribution of bug positions is uniform. While in the first RQ2 experiment we examined potential effects at scale, in this second RQ2 experiments we focus on causal analysis, isolating bug position and file size. By design, this experiment holds all potential confounding factors constant, thus isolating the bug position's impact. In other words, controlling bug position while keeping files unchanged ensured other properties remained fixed. Specifically, in our approach to constructing the experiment, we adopted a methodology consisting of three primary steps.

In the first step, for each file, we identify and isolate the main line where the vulnerability occurs. If this vulnerable line resides within a function containing fewer than 500 characters, the entire function is marked as the block of vulnerable code to be moved. Conversely, if the function exceeds 500 characters, a segment encompassing a few lines before and after the vulnerable line is extracted. This segment is then refactored into a new function of approximately 500 characters, which is the block of vulnerable code to be moved.

Let S be the size of the file, the second step involves creating $\frac{S}{500}$ new files, positioning the buggy function at different intervals: at the beginning for the first file, at the 500th character for the second, and so on, increasing by 500 characters for each subsequent file. The remaining space in each file is filled with padding characters, using either the rest of the file or contents from files in the same repository.

For the final step, we employ the 0/1 knapsack algorithm [37] to construct $\frac{S}{500}$ new files, each of size S, wherein the buggy function is positioned at different locations. The content of these new files is divided into two segments: one before the buggy function and one after it. Each padding segment is treated as a separate 'knapsack' with capacities determined by the function's position. For example, if the function is placed at position n, the first segment's capacity is $500 \times (n-1)$, and the second is $S - 500 \times n$.

To implement this, the original file content is divided into functions or independent logic units, which are then distributed into these knapsacks as padding. Each logic unit is measured in characters. To address cases where the padding does not perfectly match the knapsack's capacity constraints, we use a relaxed version of the 0/1 knapsack algorithm. Instead of requiring an exact match to the capacity, we aim for the best fit possible that is closest to the target capacity, whether it is slightly under or over. We define a tolerance value that allows the total size of selected functions to exceed the capacity by a small amount, i.e., approximately 200 characters.

The aforementioned procedure, however, presents several challenges. A primary challenge is the need to refactor the vulnerable file to allow the free movement of a block code throughout the file. This often requires a deep understanding of both the vulnerability and the underlying repository. Another significant challenge is finding suitable padding content when the target file size exceeds the actual file size. Using a LLM to address these challenges was excluded, as it could introduce significant bias into the experiment. For instance, the evaluated LLM might be already familiar with the code generated for the padding.

Given the experimental setup requires submitting hundreds of inputs to the LLM for each vulnerable file, considering all 794 vulnerable files would substantially increase both the costs and the duration of the experiment. Therefore, we needed to identify the smallest number of files necessary to achieve significant results.

To determine this, we conducted an *a priori* logistic regression power analysis [13]. This analysis considered a one-tailed test, focusing on a strong negative association between bug localization rate and bug position. It assumed a uniform distribution of the predictor, meaning each bug position in a file is equally likely. Additionally, the analysis aimed for a power $\beta \geq .8$ with a significance level (α) of .05. Under the null hypothesis, the probability of finding a bug given its position $Pr(Y = 1|X = 1)$ was set to .5. The analysis also accounted for an R^2 value of .2 for other predictors, indicating a small to moderate impact of file content. Given the results of the previous experiments, a medium to large effect size was assumed, corresponding to an odds ratio greater than 3.47 [6].

The *a priori* power analysis suggested a minimum sample size of 273 per logistic regression to achieve the desired statistical power. Since each logistic regression considers $\frac{S}{500}$ files, and the maximum S we can consider is less than 30,000 characters due to the context window limitation of Llama 3 (the LLM with the smallest context window; see Table 2), we need at least $\frac{273 \cdot 500}{\sim 30{,}000} \approx 5$ different vulnerability instances per CWE type.

As a result, we selected only 5 different instances of vulnerability per CWE type, limiting the scope to $5 \times 3 = 15$ vulnerable files instead of using the entire dataset of 794.

For the code-in-the-haystack experiment, we considered $S = \{4,000, 8,000, 16,000, 25,000\}$ characters, resulting in 530 different combinations of files and positions constructed for each CWE type and logistic regression, amounting to 28,830,000 characters in total. We chose 25,000 characters as the largest S since it is slightly lower than the maximum input size of Llama 3. This allowed us to run the same experiments across all six LLMs.

Each file was processed individually by each LLM using the same prompt and matching strategy described in Section 5. Given that all the considered LLMs are non-deterministic and that we

have (only) 5 distinct vulnerabilities for each position, file size, and CWE type, we conducted the experiment five times with each LLM to minimize variability in the results.

To evaluate the results, we assigned a score of +1 when a file's vulnerability was correctly located and -1 when it was incorrectly located. For each run, file size S, bug position n, LLM, and CWE type, we averaged the scores across the (5) different instances of vulnerabilities, resulting in an average number within the range of $[-1, 1]$. This score represents the effectiveness of the LLM at locating vulnerabilities of the given CWE type. We then averaged these results across multiple runs to achieve more reliable and consistent outcomes.

Results. Figure 3 displays the heatmap resulting from the code-in-the-haystack experiment for the top three most accurate models, as indicated in Table 3: ChatGPT 4, ChatGPT 4o, and Llama 3. Meanwhile, Figure 4 shows the logistic regressions for all models. These regressions confirm the previous results, indicating a statistically significant negative impact of bug position and file size on vulnerability detection. Interestingly, this trend is more pronounced with bug position (i.e., the logistic regressions are steeper) than with file size, suggesting that position plays a bigger role.

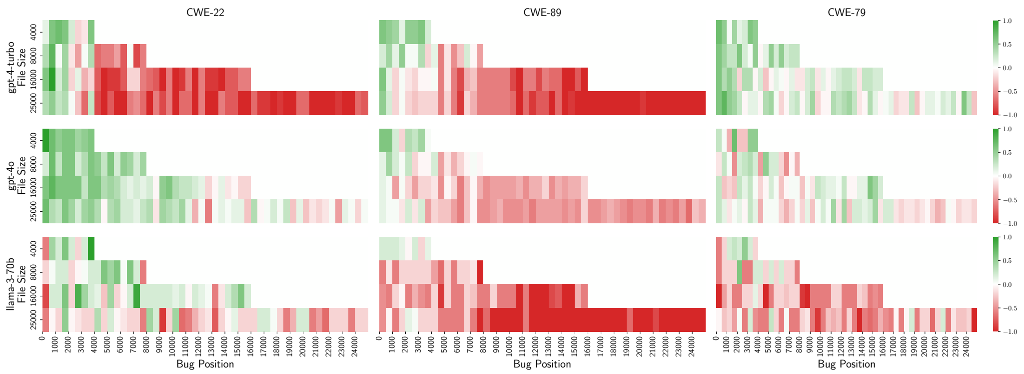

Fig. 3. Code-in-the-Haystack: Heatmap results for the most accurate LLMs.

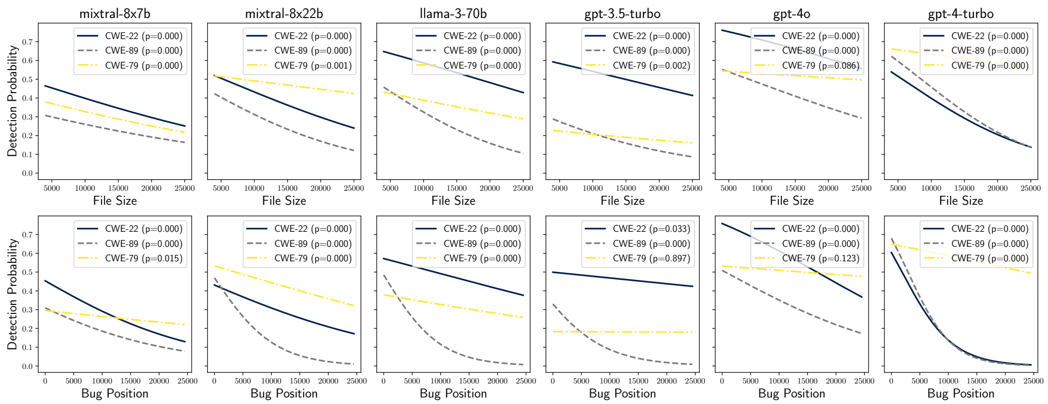

Fig. 4. Code-in-the-Haystack: Comparative logistic regression analysis.

Notably, Figure 3 shows that, when dealing with the largest files, the LLMs predominantly focus on the beginning of the file, disregarding the content towards the end of the context window.

This observation indicates that the models do not suffer from a ‘lost-in-the-middle’ phenomenon. Instead, they exhibit a ‘lost-in-the-end’ problem. The degree to which this trend occurs varies among different models and CWE types.

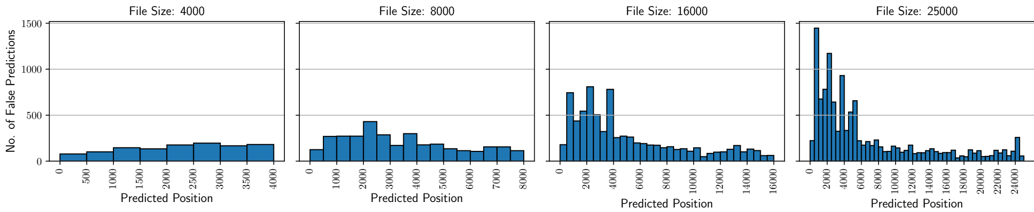

Fig. 5. Code-in-the-Haystack: Distribution of incorrectly predicted bug positions.

To further verify this finding, we conducted an error analysis. As shown in Figure 5, we plotted the distribution of incorrect bug positions predicted by all the LLMs. The results clearly indicate that the LLMs tend to predict vulnerabilities at the earliest positions (< 6,000) when they fail to identify a vulnerability, often neglecting positions towards the end of the file.

Our code-in-the-haystack experiment is designed to have a uniform distribution of bug positions for each of the file sizes S. If there were no ‘lost-in-the-end’ phenomenon, we would expect to see a more-or-less uniform distribution of errors in Figure 5, similar to what we observe for S = {4,000, 8,000}. However, for S = {16,000, 25,000}, the predicted positions are skewed towards the file’s beginning. These observations support the following finding:

Finding 2: The LLMs we examined struggle to detect security bugs—at least those of types CWE-22, CWE-89, and CWE-79—toward the end of longer files.

Figures 3 and 5 show a threshold beyond which the LLMs starts struggling with vulnerability detection. For example, ChatGPT 4, 4o, and Llama 3 have difficulty detecting CWE-89 type vulnerabilities when the input exceeds 4,000 characters. The next section explores how to identify this threshold for each CWE type and LLM.

7 RQ3: Input Size Identification

With RQ3, we aim to identify best practices for practitioners that use popular chat-based LLMs, such as GPT models, which suffer from the ‘lost-in-the-end’ issue. Since these LLMs struggle to detect vulnerabilities in files approaching their context window limits, we seek to provide guidance on the right input size for reliable detection. Specifically, we want to identify the largest input size for a CWE type at which the model’s detection capabilities have the highest recall. We hypothesize that the best input size can be determined by observing changes in the LLM’s ability to identify the exact location of a vulnerability when positioned into file chunks of different sizes. To investigate this hypothesis, we use the data from the first experiment of RQ2 (cf. Section 6.1), i.e., the 794 vulnerable files.

Methodology. Our experimental procedure involves a naive chunking strategy where the content of a file is divided into blocks of lines totaling up to a maximum of k characters. Each line is preserved in its entirety to ensure at least one chunk contained the critical line for the LLM to identify. Although simplistic, this approach demonstrates that even basic chunking could enhance the LLM’s performance.

Table 4. Chunk sizes yielding the highest recall for each LLM and CWE type. Recall improvements are compared to the baseline values from Table 3. Best values are in bold.

CWE-ID	Model	Chunk Size	Accuracy	Recall	Recall Improv.
CWE-22	mixtral-8x7b	500	.206	.428	+21.59%
	mixtral-8x22b	3000	.219	.552	+15.97%
	llama3-70b	1500	.197	.495	+15.12%
	gpt-3.5-turbo	1500	.220	.533	**+31.93%**
	gpt-4-turbo	6500	.363	.657	+9.50%
	gpt-4o	6500	.320	**.666**	+16.64%
CWE-89	mixtral-8x7b	1500	.183	.452	+40.37%
	mixtral-8x22b	500	.122	**.657**	**+52.08%**
	llama3-70b	1500	.330	.554	+15.18%
	gpt-3.5-turbo	500	.278	.636	+35.61%
	gpt-4-turbo	500	.462	.630	+35.19%
	gpt-4o	1500	.238	.650	+26.46%
CWE-79	mixtral-8x7b	500	.375	.309	+68.85%
	mixtral-8x22b	500	.147	**.428**	+81.36%
	llama3-70b	1500	.474	.278	+19.31%
	gpt-3.5-turbo	500	.397	.376	**+95.83%**
	gpt-4-turbo	500	.492	.404	+66.26%
	gpt-4o	500	.380	.407	+31.72%

Each chunk was evaluated using the same experimental prompts used in the investigations of RQ1 and RQ2. Due to the increase in negative examples compared to RQ1, direct comparisons of accuracy and F1 scores are not viable. Instead, we focused on comparing the recall metric, representing the percentage of correctly identified vulnerabilities by the LLM. The chunking strategy was implemented using k values of {6,500, 3,000, 1,500, 500}. The largest chunk size, 6,500, corresponds to approximately the median file size for CWE-89 (see Table 1). A chunk size of 3,000 represents the lower quartile of file sizes across all CWE types. The 1,500 size is slightly above the average function size, while 500 is less than the average function size, providing a finer granularity.

Results. According to the results detailed in Table 4, we have an average recall improvement of over +37% across all models and CWE types due to chunking, although this strategy led to a decrease in precision, indicated by an increase in false positives due to a greater number of chunks with no vulnerability.

For CWE-22 with a 1,500 character chunk, ChatGPT 3.5 showed a significant +31.93% recall improvement. In contrast, ChatGPT 4 (turbo) with a 6,500 character chunk reached a +9.5% improvement for the same CWE. ChatGPT 3.5 with a 500 character chunk for CWE-79 showed the most substantial improvement, almost doubling the baseline recall (+95.83%).

Interestingly, the smaller LLMs demonstrated the most significant improvements in recall and appeared to benefit more from smaller chunk sizes. For example, in CWE-89, Mixtral 8x22b, using a chunk size of 500, achieved a +52.08% improvement in recall, the highest among all models for this specific CWE type. Furthermore, non-commercial LLMs also showed greater performance gains with smaller chunks, narrowing the performance gap with commercial models. In fact, after chunking, open-source models frequently outperformed commercial ones in vulnerability detection.

The data also reveal disparities in how different CWE types respond to chunking. For instance, CWE-79 is the most susceptible to 'lost-in-the-end' issues, benefiting the most from smaller chunk sizes of 500 characters (less than a function!). CWE-89 and CWE-22 followed, showing optimal safe input sizes which are around 1,500 and 3,000-6,500 characters, respectively. When we compare these input sizes to the median file and function sizes in Table 1, we observe notable differences across CWE types. Specifically, CWE-22 requires a small-to-medium file size, whereas CWE-89 and CWE-79 require the sizes of a small function and less than half a function, respectively. These observations substantiate the following finding:

Finding 3: Chunking input files into smaller blocks enhances LLMs's recall in vulnerability detection, with optimal sizes varying a lot by CWE type.

8 Threats to Validity

In evaluating the findings, several extrinsic and intrinsic potential threats to validity need to be considered.

Extrinsic Threats. While we included a variety of popular LLMs, such as Mixtral and Llama, as well as several versions of ChatGPT, the conclusions drawn are primarily applicable to the contexts and vulnerability types explicitly tested, i.e., CWE-22, CWE-79, and CWE-89. The performance of these models might differ when applied to other CWE types or in different software contexts, such as different programming languages.

OpenAI's continuous updates to their models also pose a challenge; the accuracy and F1 scores presented might change if the experiments are rerun at a different time. Although we aimed for consistency and simplicity in the prompts used, future versions of ChatGPT may perform differently or require new types of prompts.

Lastly, our analysis using logistic regression to assess the impact of file size and vulnerability position on detection probability assumes a linear relationship, which may not capture more complex patterns adequately.

Intrinsic Threats. Generally speaking, it is impossible to mathematically guarantee that a non-trivial piece of code (e.g., a vulnerability fix) is fully bug-free. This impossibility stems from Rice's theorem [39]. Nevertheless, we are certain that the post-fix code we used for answering RQ1 resolves at least one instance of vulnerability, and that this vulnerability must be detected by the LLM. Instead, our core findings and contributions, which stem from RQ2 and RQ3, are not affected by this problem.

The algorithmic construction and manual refactoring of our dataset's files for the code-in-the-haystack experiments could have introduced syntactical errors, such as misplaced import statements. However, this is not a major concern as the study focuses on the 'lost-in-the-end' phenomenon in detecting specific types of security weaknesses rather than on syntactical errors.

The non-deterministic nature of the AI models, particularly the non-commercial ones that lack control over the random seed, adds another layer of complexity. To mitigate this, we repeated the code-in-the-haystack experiments five times and averaged the results, incurring costs exceeding 400 USD. However, due to these costs, this approach was not feasible for other experiments, and even five runs may be insufficient. Nonetheless, the consistent observation of the 'lost-in-the-end' issue across all studied LLMs and the statistical significance of the logistic regression support the reliability of our findings, reducing the likelihood that the observed trends are due to chance.

Finally, the range of chunk sizes k we considered is arbitrary, and better chunk sizes might exist. Due to the high costs of running large experiments with LLMs, we were limited to testing a few k values using a dichotomic search.

9 Discussion

The three research questions outlined in Section 3 focused on evaluating the effectiveness of several popular LLMs in detecting in-file vulnerabilities. Specifically, they examined the influence of file size and the position of the vulnerability, as well as the optimal input size for maximizing detection accuracy.

RQ1. The findings presented in Section 5 address RQ1, indicating that the performance of off-the-shelf LLMs is generally low, with accuracy scores below .4, and varies significantly across different CWE types. No model consistently outperformed others across all aspects, though commercial models generally show superior performance compared to open-source models. Notably, the majority of the LLMs were more effective in detecting CWE-22 and CWE-89 compared to CWE-79. This suggests that the complexity and commonality of vulnerabilities influence detection capabilities. Interestingly, our results diverge from previous studies [50, 51], which suggested that more frequent vulnerabilities are easier to detect for these models. Our findings show that the most frequent vulnerabilities can be the hardest to spot for LLMs.

RQ2. For RQ2, our experiments (cf. Section 6) reveal a clear pattern: as file size increases or the vulnerability is positioned towards the end of the file, the probability of its detection by LLMs decreases. This finding was consistent across all models and types of vulnerabilities studied, regardless of the LLMs' maximum context window size. This highlights the challenges LLMs face with large inputs and their sensitivity to the 'lost-in-the-end' phenomenon. Additionally, our analysis showed that different CWE types are handled differently. For example, CWE-79 had the deepest decline in vulnerability localization probability, suggesting that some vulnerabilities may require more context for accurate identification, but current LLMs limitations in handling large contexts prevent proper localization.

Notably, the distributions of bug positions and file sizes shown in Figure 2 (left) indicate that real-world bugs frequently occur within the first 10,000 characters of a file. The same bar plot also suggests a correlation between bug position and file size, possibly because smaller files inherently constrain bug placement. Indeed, the file size distribution is skewed toward smaller files, with most containing fewer than 20,000 characters. To account for this correlation, we conducted the code-in-the-haystack experiment (Section 6.2) using uniformly and independently distributed bug positions and file sizes. The results still indicate a 'lost-in-the-end' phenomenon.

Importantly, when analyzing the results of the code-in-the-haystack experiment, we observe that the distribution of incorrectly predicted bug positions for files of size S =25,000 (see Figure 5, rightmost box) closely resembles the distribution of bug positions in real-world files collected from the CVE catalog. The LLMs incorrectly predict that bug positions occur within the initial 10,000 characters of a file. This suggests that the skewed distribution of real-world bug positions may be why LLMs (likely trained on that data) tend to focus on the beginning of files, using code position as a heuristic for bug localization rather than analyzing the underlying code. This observation aligns with the findings of Nikankin et al. [33], which show that LLMs often rely on a 'bag of heuristics' to solve problems. If this heuristic reliance is accurate, alternative training strategies may be necessary. For instance, training LLMs with more uniformly distributed bug positions and file sizes or refining the training pipeline could potentially mitigate the 'lost-in-the-end' phenomenon.

RQ3. The results from our chunking experiments addressing RQ3 suggest that smaller input sizes can significantly improve the recall of LLMs in detecting vulnerabilities (up to +95%), depending on the CWE type. Specifically, smaller chunk sizes of 500 to 1,500 characters were more effective for the most frequent CWE types, while larger chunk sizes up to 6,500 characters or more were most effective for the least frequent CWE type, CWE-22. This indicates that at least for CWE-22, in-file vulnerability detection is needed.

Given that the vulnerabilities we are studying rank among the top 10 most dangerous ones (cf. Section 4), maximizing recall (i.e., the true positive rate) can be more important than precision or accuracy. Therefore, adopting a chunking strategy like the one described could, with little to no additional cost, increase the vulnerability detection rates of state-of-the-art LLMs, effectively mitigating their 'lost-in-the-end' issues.

Nonetheless, chunking is a useful but temporary solution for managing large files, often leading to increased alert fatigue—a problem also identified by Hassan et al. [19]—due to a rise in false positives. When examining this phenomenon, we find that most false positives result from a loss of context caused by chunking. For example, all LLMs incorrectly flag the following as buggy: 'query = "SELECT * FROM users WHERE username="+user_input+""' simply because the 'user_input' sanitization is out of context.

A potential solution to this problem is 'smarter chunking', where variable definitions and manipulations are included in the chunk. However, this chunking approach could effectively address only input sanitization issues such as SQL injection, path traversal, and XSS. But, given that these are among the most dangerous and common vulnerabilities, employing this specialized strategy may still be worthwhile. Instead, for other types of vulnerabilities, a more effective approach is to address the 'lost in the end' issue directly within the LLM architecture and training pipeline.

Although the chunking strategy employed is somewhat naive and encounters similar issues as function-level detection due to limited contextual information, the results (see Table 4) indicate that the most effective chunk sizes can exceed the average function size (shown in Table 1). This further confirms that function-level detection may be ineffective, particularly for certain vulnerabilities like CWE-22 and, to a lesser extent, CWE-89.

Interestingly, when considering smaller input sizes, the performance gap between commercial and open-source models diminished notably. In some cases, open-source models performed on par with or even better than commercial ones, suggesting that the main advantage of commercial models may lie in a better handling of context windows. This indicates that advancements in handling context windows could significantly improve the existing technology for vulnerability detection. However, it is unclear whether these LLMs suffer from the 'lost-in-the-end' problem because they had mostly seen data where vulnerability detection is done predominantly on function-level data.

While our findings highlight significant improvements in recall through chunking, they also underscore the need for practitioners to adopt a more nuanced approach when selecting and configuring LLMs for vulnerability detection. Practitioners should consider not only the chunk size but also the nature of the CWE type and the model's context-handling capabilities. For instance, smaller chunk sizes (500–1,500 characters) yield superior results for CWE types such as CWE-79 and CWE-89, whereas larger chunks (up to 6,500 characters) may be necessary for more context-dependent vulnerabilities like CWE-22. Commercial models like GPT-4-turbo and GPT-4o demonstrate stronger context retention, making them more effective for CWE-22. Instead, open-source models such as Mixtral-8x22b can achieve higher recall performance when smaller context retention is required (e.g., for CWE-79 and CWE-89), outperforming their commercial counterparts, which appear to be more specialized for longer contexts.

Hypotheses. We hypothesize that the problem is the combination of the context-dependency of certain vulnerabilities and the 'lost-in-the-end' issue, so that CWE types requiring more context generally perform worse. For instance, detecting XSS (CWE-79) requires understanding how user inputs are handled, sanitized, and embedded into web pages, which can span various functions, files, or even different programming languages (e.g., JavaScript, HTML). If the model cannot capture the broader context (maybe because it has not seen many large contexts during training), it might miss the vulnerability, as shown by our empirical results.

Similarly, if the code context is not correctly captured by the model, it may mistakenly identify a non-buggy line as having a vulnerability due to the fix being elsewhere in the file. Hence, contrary to what [51] suggest, it is possible that the harder-to-locate bugs are those with greater variability in the type and quantity of code modifications necessitated by a security patch, rather than the rarer vulnerabilities, i.e., the tail CWE types.

Another possibility is that the code used for training these LLMs frequently contains unidentified bugs, leading the models to replicate them or fail to recognize them as bugs. Indeed, it is possible that the most frequently fixed CWE vulnerability types (CWE-79) also more frequently appear unfixed in production or open-source code. If the model cannot handle a large context effectively, it might learn these vulnerabilities as normal non-buggy code, as suggested by the experiments conducted by [18], which indicate that LLMs tend to replicate bugs.

Future Work. Future work should verify whether any of the hypotheses mentioned above actually hold. Moreover, given that code, unlike natural language, does not follow a linear, top-down format, future research should explore more advanced architectures for these LLMs. This could potentially include permutation-invariant neural networks, such as graph neural networks, where reordering a function's position does not affect the model's ability to detect vulnerabilities.

10 Conclusion

This paper characterized and highlighted the lost-in-the-end issue in popular chat-based LLMs. Recognizing this issue is important for software developers and engineers who extensively use these specific LLMs for coding and debugging activities such as file-level vulnerability detection.

In this study, we studied the effectiveness and limitations of popular chat-based LLMs in detecting vulnerabilities within entire source code files, focusing on three of the most dangerous and common CWE vulnerability types: CWE-22, CWE-89, and CWE-79. Our findings reveal significant variability in LLM performance across these vulnerabilities and highlight a new challenge: the 'lost-in-the-end' issue, where vulnerabilities located towards the end of files are less likely to be detected.

We showed that the 'lost-in-the-end' issue can considerably affect the performance of LLMs on file-level vulnerability detection. This also implies that the same issue could affect LLMs at similar tasks involving reasoning over large software files such as code review automation, generic bug localization, code summarization.

The implications of our findings are twofold. Firstly, they suggest that further improvements in LLMs are needed before they can be reliably used for vulnerability detection in software development. This includes enhancements in their ability to handle large inputs and in their sensitivity to the placement of vulnerabilities within files. Secondly, our study highlights the potential of simple yet effective strategies like input chunking to significantly enhance the performance of existing LLMs, which could be readily applied in current software development practices.

Although our analysis (Figure 2) indicates that the smallest files are the most common, the 'lost-in-the-end' issue is still relevant because the largest files should not be ignored and "the devil is in the tails" [50]. Therefore, to eliminate the most dangerous vulnerabilities, we strongly believe that future research should keep increasing the context window of these LLMs up to the repository level, but in a way that is more robust to the 'lost-in-the-end' issue.

Data Availability

All the data and scripts used for this paper are available in our replication package [2].

Acknowledgments

F. Sovrano and A. Bacchelli gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Project 200021_197227.

References

[1] Shengnan An, Zexiong Ma, Zeqi Lin, Nanning Zheng, and Jian-Guang Lou. 2024. Make Your LLM Fully Utilize the Context. *CoRR* abs/2404.16811 (2024). doi:10.48550/ARXIV.2404.16811 arXiv:2404.16811

[2] Adam Bauer and Francesco Sovrano. 2025. Replication Package for In-file Vulnerability Detection. https://doi.org/10.5281/zenodo.14840519.

[3] Dipkamal Bhusal, Md Tanvirul Alam, Le Nguyen, Ashim Mahara, Zachary Lightcap, Rodney Frazier, Romy Fieblinger, Grace Long Torales, and Nidhi Rastogi. 2024. SECURE: Benchmarking Generative Large Language Models for Cybersecurity Advisory. *CoRR* abs/2405.20441 (2024). doi:10.48550/ARXIV.2405.20441 arXiv:2405.20441

[4] Alessio Buscemi. 2023. A Comparative Study of Code Generation using ChatGPT 3.5 across 10 Programming Languages. *CoRR* abs/2308.04477 (2023). doi:10.48550/ARXIV.2308.04477 arXiv:2308.04477

[5] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. 2023. When ChatGPT Meets Smart Contract Vulnerability Detection: How Far Are We? doi:10.48550/ARXIV.2309.05520 arXiv:2309.05520

[6] Henian Chen, Patricia Cohen, and Sophie Chen. 2010. How Big is a Big Odds Ratio? Interpreting the Magnitudes of Odds Ratios in Epidemiological Studies. *Commun. Stat. Simul. Comput.* 39, 4 (2010), 860–864. doi:10.1080/03610911003650383

[7] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023. Extending Context Window of Large Language Models via Positional Interpolation. *CoRR* abs/2306.15595 (2023). doi:10.48550/ARXIV.2306.15595 arXiv:2306.15595

[8] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David A. Wagner. 2023. 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, RAID 2023, Hong Kong, China, October 16-18, 2023*. ACM, 654–668. doi:10.1145/3607199.3607242

[9] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. 2023. Evaluation of ChatGPT Model for Vulnerability Detection. *CoRR* abs/2304.07232 (2023). doi:10.48550/ARXIV.2304.07232 arXiv:2304.07232

[10] Alfred DeMaris and Steven H Selman. 2013. *Converting data into evidence: A statistics primer for the medical practitioner.* Springer. doi:10.1007/978-1-4614-7792-1

[11] Aparna Dhinakaran. 2024. The Needle in a Haystack Test. https://towardsdatascience.com/the-needle-in-a-haystack-test-a94974c1ad38. Accessed: 2024-06-03.

[12] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. 2024. LongRoPE: Extending LLM Context Window Beyond 2 Million Tokens. *CoRR* abs/2402.13753 (2024). doi:10.48550/ARXIV.2402.13753 arXiv:2402.13753

[13] Edgar Erdfelder, Franz Faul, and Axel Buchner. 1996. GPOWER: A general power analysis program. *Behavior research methods, instruments, & computers* 28 (1996), 1–11. doi:10.3758/BF03203630

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. doi:10.18653/V1/2020.FINDINGS-EMNLP.139

[15] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 608–620. doi:10.1145/3524842.3528452

[16] Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. 2023. ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We?. In *30th Asia-Pacific Software Engineering Conference, APSEC 2023, Seoul, Republic of Korea, December 4-7, 2023*. IEEE, 632–636. doi:10.1109/APSEC60848.2023.00085

[17] GitLab. 2022. 2022 DevSecOps Report. https://learn.gitlab.com/dev-survey-22/2022-devsecops-report. Accessed: 2024-06-03.

[18] Sivana Hamer, Marcelo d'Amorim, and Laurie Williams. 2024. Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers. doi:10.48550/ARXIV.2403.15600 arXiv:2403.15600

[19] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. 2019. NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/nodoze-combatting-threat-alert-fatigue-with-automated-provenance-triage/

[20] Wenpin Hou and Zhicheng Ji. 2024. Comparing large language models and human programmers for generating programming code. *Advanced Science* (2024), 2412279. doi:10.1002/advs.202412279

[21] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models. doi:10.48550/ARXIV.2402.14848 arXiv:2402.14848

[22] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. SnapKV: LLM Knows What You are Looking for Before Generation. doi:10.48550/ARXIV.2404.14469 arXiv:2404.14469

[23] Chongyang Liu, Xiang Chen, Xiangwei Li, and Yinxing Xue. 2024. Making vulnerability prediction more practical: Prediction, categorization, and localization. *Information and Software Technology* 171 (2024), 107458. doi:10.1016/j.infsof.2024.107458

[24] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Trans. Assoc. Comput. Linguistics* 12 (2024), 157–173. doi:10.1162/TACL_A_00638

[25] Yu Liu, Lang Gao, Mingxin Yang, Yu Xie, Ping Chen, Xiaojin Zhang, and Wei Chen. 2024. VulDetectBench: Evaluating the Deep Capability of Vulnerability Detection with Large Language Models. *CoRR* abs/2406.07595 (2024). doi:10.48550/ARXIV.2406.07595 arXiv:2406.07595

[26] Daniel Machlab and Rick Battle. 2024. LLM In-Context Recall is Prompt Dependent. doi:10.48550/ARXIV.2404.08865 arXiv:2404.08865

[27] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. 2023. Prompt Engineering in Large Language Models. In *International Conference on Data Intelligence and Cognitive Informatics*. Springer, 387–402. doi:10.1007/978-981-99-7962-2_30

[28] MITRE. 2024. Common Vulnerabilities and Exposures (CVE) Catalog. https://www.cve.org. Accessed: 2024-06-03.

[29] MITRE Corporation. 2024. 2024 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html. [Online; accessed 5-February-2025].

[30] Daye Nam, Andrew Macvean, Vincent J. Hellendoorn, Bogdan Vasilescu, and Brad A. Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 97:1–97:13. doi:10.1145/3597503.3639187

[31] Kollin Napier, Tanmay Bhowmik, and Shaowei Wang. 2023. An empirical study of text-based machine learning models for vulnerability detection. *Empir. Softw. Eng.* 28, 2 (2023), 38. doi:10.1007/S10664-022-10276-6

[32] Thu-Trang Nguyen and Hieu Dinh Vo. 2024. Context-based statement-level vulnerability localization. *Inf. Softw. Technol.* 169 (2024), 107406. doi:10.1016/J.INFSOF.2024.107406

[33] Yaniv Nikankin, Anja Reusch, Aaron Mueller, and Yonatan Belinkov. 2024. Arithmetic Without Algorithms: Language Models Solve Math With a Bag of Heuristics. *CoRR* abs/2410.21272 (2024). doi:10.48550/ARXIV.2410.21272 arXiv:2410.21272

[34] Arka Pal, Deep Karkhanis, Manley Roberts, Samuel Dooley, Arvind Sundararajan, and Siddartha Naidu. 2023. Giraffe: Adventures in Expanding Context Lengths in LLMs. doi:10.48550/ARXIV.2308.10882 arXiv:2308.10882

[35] Chao-Ying Joanne Peng, Kuk Lida Lee, and Gary M Ingersoll. 2002. An introduction to logistic regression analysis and reporting. *The journal of educational research* 96, 1 (2002), 3–14. doi:10.1080/00220670209598786

[36] Tao Peng, Shixu Chen, Fei Zhu, Junwei Tang, Junping Liu, and Xinrong Hu. 2023. PTLVD:Program Slicing and Transformer-based Line-level Vulnerability Detection System. In *23rd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2023, Bogotá, Colombia, October 2-3, 2023*, Leon Moonen, Christian D. Newman, and Alessandra Gorla (Eds.). IEEE, 162–173. doi:10.1109/SCAM59687.2023.00026

[37] David Pisinger. 1997. A Minimal Algorithm for the 0-1 Knapsack Problem. *Oper. Res.* 45, 5 (1997), 758–767. doi:10.1287/OPRE.45.5.758

[38] Moumita Das Purba, Arpita Ghosh, Benjamin J. Radford, and Bill Chu. 2023. Software Vulnerability Detection using Large Language Models. In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023 - Workshops, Florence, Italy, October 9-12, 2023*. IEEE, 112–119. doi:10.1109/ISSREW60843.2023.00058

[39] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* 74, 2 (1953), 358–366. doi:10.2307/1990888

[40] Sonatype. 2023. State of the Software Supply Chain 9th Edition. https://www.sonatype.com/state-of-the-software-supply-chain/introduction. Accessed: 2024-06-03.

[41] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T. Barr, and Wei Le. 2024. A Comprehensive Study of the Capabilities of Large Language Models for Vulnerability Detection. (2024). doi:10.48550/ARXIV.2403.17218 arXiv:2403.17218

[42] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-Based Language Models for Software Vulnerability Detection. In *Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022*. ACM, 481–496. doi:10.1145/3564625.3567985

[43] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse K. Coskun, and Gianluca Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 862–880. doi:10.1109/SP54263.2024.00210

[44] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 38–49. doi:10.1109/SANER.2018.8330195

[45] Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. 2024. ReposVul: A Repository-Level High-Quality Vulnerability Dataset. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 472–483. doi:10.1145/3639478.3647634

[46] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. doi:10.18653/V1/2021.EMNLP-MAIN.685

[47] Peng Xu, Wei Ping, Xianchao Wu, Lawrence McAfee, Chen Zhu, Zihan Liu, Sandeep Subramanian, Evelina Bakhturina, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Retrieval meets Long Context Large Language Models. doi:10.48550/ARXIV.2310.03025 arXiv:2310.03025

[48] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2023. Prompt-Enhanced Software Vulnerability Detection Using ChatGPT. doi:10.48550/ARXIV.2308.12697 arXiv:2308.12697

[49] Peitian Zhang, Zheng Liu, Shitao Xiao, Ninglu Shao, Qiwei Ye, and Zhicheng Dou. 2024. Soaring from 4K to 400K: Extending LLM's Context with Activation Beacon. doi:10.48550/ARXIV.2401.03462 arXiv:2401.03462

[50] Xin Zhou, Kisub Kim, Bowen Xu, Jiakun Liu, DongGyun Han, and David Lo. 2023. The Devil is in the Tails: How Long-Tailed Code Distributions Impact Large Language Models. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 40–52. doi:10.1109/ASE56229.2023.00157

[51] Xin Zhou, Ting Zhang, and David Lo. 2024. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. doi:10.48550/ARXIV.2401.15468 arXiv:2401.15468

[52] Xin Zhou, Ting Zhang, and David Lo. 2024. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 47–51. doi:10.1145/3639476.3639762

Received 2024-09-09; accepted 2025-01-14