Leveraging Large Language Models to Detect npm Malicious Packages

Nusrat Zahan*, Philipp Burckhardt†, Mikola Lysenko†, Feross Aboukhadijeh†, Laurie Williams*
*North Carolina State University
Email: [nzahan, lawilli3]@ncsu.edu
†Socket, Inc
Email: [philipp, mik, feross]@socket.dev

2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE) | 979-8-3315-0569-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICSE55347.2025.00146

***Abstract*—Existing malicious code detection techniques demand the integration of multiple tools to detect different malware patterns, often suffering from high misclassification rates. Therefore, malicious code detection techniques could be enhanced by adopting advanced, more automated approaches to achieve high accuracy and a low misclassification rate. *The goal of this study is to aid security analysts in detecting malicious packages by empirically studying the effectiveness of Large Language Models (LLMs) in detecting malicious code.* We present SocketAI, a malicious code review workflow to detect malicious code. To evaluate the effectiveness SocketAI, we leverage a benchmark dataset of 5,115 npm packages, of which 2,180 packages have malicious code. We conducted a baseline comparison of GPT-3 and GPT-4 models with the state-of-the-art CodeQL static analysis tool, using 39 custom CodeQL rules developed in prior research to detect malicious Javascript code. We also compare the effectiveness of static analysis as a pre-screener with SocketAI workflow, measuring the number of files that need to be analyzed and the associated costs. Additionally, we performed a qualitative study to understand the types of malicious packages detected or missed by our workflow. Our baseline comparison demonstrates a 16% and 9% improvement over static analysis in precision and F1 scores, respectively. GPT-4 achieves higher accuracy with 99% precision and 97% F1 scores, while GPT-3 offers a more cost-effective balance at 91% precision and 94% F1 scores. Pre-screening files with a static analyzer reduces the number of files requiring LLM analysis by 77.9% and decreases costs by 60.9% for GPT-3 and 76.1% for GPT-4. Our qualitative analysis identified data theft, execution of arbitrary code, and suspicious domain categories as the top detected malicious packages.**

I. Introduction

Open-source package repositories are often misused for software supply chain attacks [23]. Attackers attempt to execute supply chain attacks by injecting malicious code into existing packages or by creating new infected packages and tricking users into downloading them [23], [32]. Classifying a package as malicious is challenging because malicious code detection necessitates comprehending the reasoning about the possible purpose and intent of the code. Given the increasing rate of software supply chain attacks, the demand for approaches to detect such attacks is also increasing.

The current malicious package detection [12], [14], [31], [38], [39], [44], [45], [47], [56] including static, rules, heuristics, differential analysis, and different machine learning techniques encounter challenges of incomplete solutions, high misclassification rates [33]. Additionally, no single technique can comprehensively address different attack types, like detecting malicious code injection, dependency confusion, and typosquat, and often fails to support languages like TypeScript and CoffeeScript or research packages [33]. As a result, practitioners need to integrate multiple tools to detect malware patterns. An advanced detection technique is needed to identify diverse attack types effectively and minimize manual review efforts with minimal misclassification. LLMs have the potential to address these limitations by detecting diverse attacks through their advanced understanding of context, semantics, and patterns, thereby reducing reliance on multiple tools.

While the existing literature lacks the contribution of using LLMs to detect malicious code, organizations are adopting LLMs as an advanced technique for threat analysis [3], [27]. However, assessing the performance of LLMs in production presents challenges: 1) comprehensive evaluation in a production environment is resource-intensive: 2) LLM missing actual malicious code (false negative); 3) validation due to the lack of ground truth; and 4) after reporting potential malicious packages to the package registry, the package managers often only acknowledge receipt of the reports without confirming the presence of malicious code. Hence, we conduct an empirical study to evaluate LLMs' performance in malicious code detection. *The goal of this study is to aid security analysts in detecting malicious packages by empirically studying the effectiveness of Large Language Models (LLMs) in detecting malicious code.* In this study, we address the following research questions (RQs)—

- **RQ1**: What performance can be achieved in malicious code detection using LLMs compared with static analysis (baseline comparison), in terms of precision, recall, and F1 score?
- **RQ2**: What is the effectiveness of using a static analyzer as a pre-screener with SocketAI in terms of the number of files that need to be analyzed and the associated cost?
- **RQ3**: What types of malicious packages are detected or missed by LLMs?

To the best of our knowledge SocketAI, a malicious code review workflow is a first-ever study leveraging LLMs to detect malicious code and compares the effectiveness of the workflow with static analysis tools. The workflow leverages Iterative Self-Refinement [28], LLM-as-a-Judge [58] and Zero-Shot-Role-Play-Chain of Thought (CoT) prompting [20], [21] techniques in ChatGPT models (GPT-3 and GPT-4). We leveraged MalwareBench [55], a benchmark dataset, and studied 5,115 npm packages, including 2,180 malicious and 2,935 neutral packages, to evaluate the performance of LLMs.

To answer RQ1, The paper presents a baseline comparison between ChatGPT models and the mature, state-of-the-

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00146

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 28,2025 at 15:40:52 UTC from IEEE Xplore. Restrictions apply.

art commercial tool, CodeQL. We used 39 custom rules of CodeQL that have been used in prior study [11] to detect malicious Javascript packages. We compare results using the standard machine learning (ML) evaluation metrics, precision, recall, and F1 score. We answer RQ2 by comparing the effectiveness of using "SocketAI only" versus using "CodeQL as a pre-screener with SocketAI " based on the number of files analyzed and the associated costs. For RQ3, we provide qualitative insights by manually evaluating the model's responses to understand the types of malicious packages identified or missed by ChatGPT and demonstrate the opportunities and strengths of these models. For the benefit of other researchers, we further report issues we encountered while using ChatGPT. Our contributions are:

- SocketAI, a malicious code review workflow to detect malicious packages in the npm ecosystem;
- Malware-domain-specific prompt to trigger LLMs reasoning in detecting malicious code;
- A baseline comparison of SocketAI with CodeQL tool to understand the effectiveness of LLMs; and
- Strengths and shortcomings of ChatGPT for large-scale code review.

This paper is organized as follows: Section II includes our design approach, discussing our dataset, allocated cost, package selection, and model selection criteria. Section III covers our proposed workflow of SocketAI. We answer our three RQs in section IV, section V, and section VI. Then, we discussed challenges in using LLM in section VII-B. The related work section is covered in section VIII. We close with limitations and discussion of our work (Section VII-C and IX).

II. Dataset and LLM Selection

Section II-A discusses the benchmark dataset used in this study, Section II-B discusses LLMs selection process, and Section II-D explains our package selection criteria with cost constraint discussed in Section II-C.

A. Benchmark Dataset: MalwareBench

Recent studies [9], [15], [22], [32] constructed malicious package datasets. However, these studies did not incorporate neutral package data into their labeled datasets. Neutral packages are needed to improve the accuracy of detection techniques. To address the lack of labeled neutral packages, Zahan et al. constructed MalwareBench [55], a labeled benchmark dataset of malicious and neutral npm packages. Zahan et al. refer to a package as *neutral* when it has no discovered malicious code [55]. The ground-truth labels for the benchmark were established using a hybrid approach: researchers manually reviewed and confirmed all malicious cases flagged by automated tools, while neutral cases were initially identified through automation and further validated via manual analysis of randomly selected samples. MalwareBench contains 3,523 malicious and 10,691 neutral npm packages. The malicious category comprises real-world malicious packages, while the neutral category includes popular, new, and randomly annotated real-world npm packages.

B. Large Language Model Selection

LLMs based on the GPT architecture have demonstrated superior performance in inference tasks compared to BERT-style models [26], [37], [51], [59]. Inference tasks involve using a model's trained knowledge with new data to generate conclusions. While other GPT-style language models exist, ChatGPT has shown noteworthy capabilities in zero-shot learning through effective prompting and in-context learning [51]. In this study, we employed the `gpt-3.5-turbo-1106` (GPT-3) and `gpt-4-1106-preview` (GPT-4) models of ChatGPT to evaluate LLMs' performance in detecting malicious packages using zero-shot learning techniques.

C. Cost Allocation

Due to the cost and time associated with using GPT models, we first allocated a budget for this project. The budget will inform the scope of our analysis regarding the number of files or packages that can be assessed. We set up a budget of $3000 because increasing the budget would extend the model's computation time and the time needed for authors to manually review the model's generated response. We conducted a preliminary analysis of 1000 Javascript files using both GPT-3 and GPT-4 following the workflow mentioned in Section IV. We found that analyzing 1000 Javascript files with GPT-3 incurred a cost of $10.5, while the same analysis with GPT-4 cost $130. Armed with this cost information, we deduced that with a budget of $3000, we could analyze around 20,000 files.

D. Package Selection

In this work, we evaluated all files within a package rather than only analyzing files containing malicious code. The approach was taken for two reasons: to address the lack of file level granularity in MalwareBench [55] and to understand false-positive and false-negative rates in our baseline comparison comprehensively. The dataset comprised 14,214 npm packages with 1,048,576 individual files. Packages varied in size, with some containing up to 60,000 files and occupying 600 MB. Analyzing exceptionally large packages could exceed our budget and may not provide sufficient diversity of malicious or neutral package types. Therefore, we excluded large packages to stay within budget and ensure diversity. Using stratified inverse random sampling [25], we selected a representative sample of malicious and neutral categories, favoring packages below the 75th percentile. The 75th quartile encompasses packages with a maximum of 25 files or a package size of 175 KB. After applying the sampling method and adhering to budget constraints (section II-C), our final dataset comprised 2,776 malicious versions (2,180 packages) and 2,974 neutral versions (2,935 packages), containing 20,926 (18,754 unique files) files.

III. SocketAI

The section discusses the multi-stage decision-maker workflow of SocketAI. Socket has been using SocketAI to detect malicious packages in production for npm, PyPI, Go, and Maven ecosystems. On average, SocketAI is used to evaluate

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 28,2025 at 15:40:52 UTC from IEEE Xplore. Restrictions apply.

TABLE I: **System Role Prompt of Initial Report**

Role: System	**Chain of Thought Prompting**
Task (D1)	As SecureGPT, a JavaScript cybersecurity analyst, your task is to review open-source dependencies in client and server-side JavaScript code for potentially malicious behavior or sabotage. This code review is specifically for JavaScript libraries that are part of larger projects and published on public package managers such as npm. Review this code for supply chain security attacks, malicious behavior, and other security risks. Keep in mind the following: Analyze code for JavaScript security issues such as code injection, data leakage, insecure use of environment variables, unsafe SQL, and random number generation. Do NOT alert on minified code that is a result of standard minification processes using tools like UglifyJS or Terser. Third-party library usage is not, by itself, suspicious behavior.
Guidelines (D2)	Spot anomalies: hard-coded credentials, backdoors, unusual behaviors, or malicious code. Watch out for malicious privacy violations, credential theft, and information leaks. Note observations about the code. Evaluate the provided file only. Indicate low confidence if more info is needed. Avoid false positives and unnecessary warnings. Keep signals and reports succinct and clear. Consider user intent and the threat model when reasoning about signals. Focus on suspicious parts of the code.
Malware (D3)	In the context of an npm package, malware refers to any code intentionally included in the package that is designed to harm, disrupt, or perform unauthorized actions on the system where the package is installed.
Malware Example (D4):	Sending System Data Over the Network, Connecting to suspicious domains, Damaging system files, Mining cryptocurrency without consent, Reverse shells, Data theft (clipboard, env vars, etc), Hidden backdoors.
Security risks (D5):	Hardcoded credentials, Security mistakes, SQL injection, DO NOT speculate about vulnerabilities outside this module.
Obfuscated code (D6):	Uncommon language features, Unnecessary dynamic execution, Misleading variables, DO NOT report minified code as obfuscated.
Malware score (D7):	0: No malicious intent, 0 - 0.25: Low possibility of malicious intent, 0.25 - 0.5: Possibly malicious behavior, 0.5 - 0.75: Likely malicious behavior, e.g., tracking scripts, 0.75 - 1: High probability of malicious behavior; do not use.
Security risk score (D8):	0 - 0.25: No significant threat; we can safely ignore, 0.25 - 0.5: Security warning, no immediate danger, 0.5 - 0.75: Security alert should be reviewed, 0.75 - 1: Extremely dangerous, package should not be used.
Confidence Score (D9)	Rate your confidence in your conclusion about whether the code is obfuscated, whether it contains malware, and the overall security risk on a scale from 0 to 1.
Code Review (D10):	Please consider both the content of the code as well as the structure and format when determining the risks. Your analysis should include the following steps: **Identify sources**: These are places where the code reads input or data. **Identify sinks**: These are places where untrusted data can lead to potential security vulnerabilities. **Identify flows**: These are source-to-sink paths. **Identify anomalies**: These are places where there is unusual code, hardcoded secrets, etc. **Conclusion**: Finally, form the conclusion of the code, provide a succinct explanation of your reasoning.
JSON Response (D11)	**Only respond in this format:** ["**purpose**":"Purpose of this source code", "**sources**":"Places where code reads input or data", "**sinks**":"Places where untrusted data can lead to potential data leak or effect", "**flows**":"Source-to-sink paths","**anomalies**":"Places where code does anything unusual", "**analysis**":"Step-by-step analysis of the entire code fragment.", "**conclusion**":"Conclusions and short summary of your findings", "**confidence**": 0-1, "**obfuscated**": 0-1, "**malware**": 0-1, "**securityRisk**": 0-1] ONLY RESPOND IN JSON. No non-JSON text responses. Work through this step-by-step to ensure accuracy in the evaluation.

TABLE II: **System Role Prompt of Critical Reports**

Role:System	**Chain of Thought Prompting**
Task	Your role is a security researcher who will review X reports of a supply chain security incident in an open source dependency. Here's your step-by-step guide: - Confirm if the reported issues are present in the code. If not, make a note. - Identify and record any errors, flaws or mistakes in the report's logic, reasoning or summary. - Review the scores given to each issue. Suggest any changes if they seem unreasonable. - Justify any risk score higher than 0.5, as it's considered serious. - Pay close attention to the malware score. If a package is doing something malicious, it must be reflected in this score. - Challenge any claims about potential vulnerabilities that are based on assumptions. - Make sure the scores are consistent with the report.
Note	Be critical. After reading all reports, give your own estimate for what you think the malware, obfuscated, and risk scores should be. Let's carefully work this out step-by-step to be sure we have all the right answers.

17 million package versions quarterly in production. As of 2024, three authors are actively developing and maintaining the workflow. In Section III-A, and III-B, we discuss our design decisions and the prompting techniques, and section

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 28,2025 at 15:40:52 UTC from IEEE Xplore. Restrictions apply.

TABLE III: **System Role Prompt of Final Report**

Role:System	Chain of Thought Prompting
Task	As an experienced JavaScript cybersecurity analyst, review X software supply chain security reports of an open source dependency. Select the best report and improve it. If all are unsatisfactory, create a new, improved summary. System Role Prompt of Initial Reports from Table I.
Note	In your final report: - Omit irrelevant signals from your report. - If you are certain code is really malicious set the malware score to 1. -ENSURE THAT YOUR FINAL SCORE IS CONSISTENT WITH THE REPORT. **ONLY RESPOND IN JSON. PLEASE DO NOT USE ANY TEXT BESIDES THE JSON REQUIRED FOR THE FINAL FORMATTED REPORT.** Let's work this out in a step-by-step way to be sure we get the correct result.

III-C discusses our proposed SocketAI workflow.

A. *Leveraging LLM-Based Techniques*

Since SocketAI is used in production to review different ecosystems, we designed our domain-specific prompt to be model-agnostic and adaptable to open and closed-source models where the prompt techniques are fundamental to any LLM. Our prompting techniques were inspired by advanced LLM-based techniques by [20], [28], [48], [52], [58], while also prioritizing a domain-targeted prompt using the author's domain knowledge. Prior studies [5], [49], [50] demonstrate that augmenting the model with domain-specific information improves the model's performance.

The SocketAI workflow employs an iterative self-refinement [28] approach, systematically leveraging LLM's responses in a sequential analytical workflow. Iterative self-refinement is a process that involves creating an initial draft response generated by the model and subsequently refining the response based on the model's self-provided feedback [28], [57]. Madaan et al. [28] showed that iterative self-refinement prompting had superior results compared to one-step response generation and improved performance (20%). Then, we also leveraged the LLM-as-a-Judge approach [58] to evaluate and score the outputs against specific criteria. Self-refinement is a loop of generation and improvement, while LLM-as-a-Judge is better for scoring or verifying outputs against criteria. We used the Zero-Shot-CoT [20], [48], [52] technique along with Role-Play prompting [21] to enhance model reasoning in evaluating npm packages. Prior studies [20], [21], [52] showed that Zero-Shot-CoT and Role-Play prompting enhance model performance across multiple domains. Subsequently, in Role-Play prompting, the role provides context about the model's identity and background, which serves as an implicit CoT trigger, thereby improving the quality of reasoning.

B. *Prompt Design*

We designed a two-fold prompt structure: the `system role` prompt serves as the instructions to set the context for how the model should behave. We assign different `system roles` for LLMs to follow in multiple steps (Section III-C) such as *SecureGPT, a JavaScript cybersecurity analyst* and *security researcher*, to narrow the focus to JavaScript's particular security issues. The `user role` prompt comprises the input content (npm packages files) we want the models to review. For each file, we include the code as a user prompt, following the prompt— *Analyze the above code for malicious behavior. Remember to respond in the required JSON format. Consider ALL of the code carefully. Check the beginning, middle, and end of the code. Work step-by-step to get the right answer.* Table I, Table II and Table III contain the prompts used in this study.

Task & Guidelines. We first augment the prompt by defining the task of reviewing JavaScript code for supply chain attacks, malicious behavior, and other security risks. Then, we provide guidelines to spot anomalies, such as hardcoded credentials, backdoors, and unusual behaviors, while noting code observations and avoiding unnecessary warnings.

Malware, Security Risk, and Obfuscated Code. We provide definitions and examples for malware, security risks, and obfuscated code. The targeted information helps the model recognize and differentiate between malware and other security vulnerabilities. Additionally, in malware detection, one of the primary challenges is prioritizing and differentiation between malware cases and security vulnerabilities. Motivated by LLM-as-a-Judge approach [58], we address this challenge by incorporating scoring metrics— `malware intent`, `security risk`, and `confidence` — to standardize the evaluation output, enabling the model to quantify the severity of findings and assisting human reviewers in prioritizing potentially malicious files. Since attackers commonly use `obfuscation` to avoid detection, it is included to identify such cases. The confidence score also aids in evaluating the reliability of the model's findings, while malware, security risk, and obfuscation are included to understand the nature of the code. Each scoring is defined in a range of 0-1, which we divide into different groups. Table I (D7-D9) are examples of scoring prompts.

Requirement: Code Review & JSON Response. Motivated by [5], [49], [50], we included code-review prompts (Table I-D10) that guide the model in step-by-step thinking and generating conclusions by detecting key identifiers of code such as `sources`, `sinks`, `flows`, `anomalies`. Additionally, large-scale analysis in production requires machine-readable reports. Therefore, we improved the prompting technique by instructing the model to conduct a sequential analysis (`purpose-sources-sinks-flows-analysis-conclusion`) and to generate the output report via a JSON format (Table I-D11). Besides the report being machine-readable, following such templates forces the model to rethink its evaluation and

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 28,2025 at 15:40:52 UTC from IEEE Xplore. Restrictions apply.

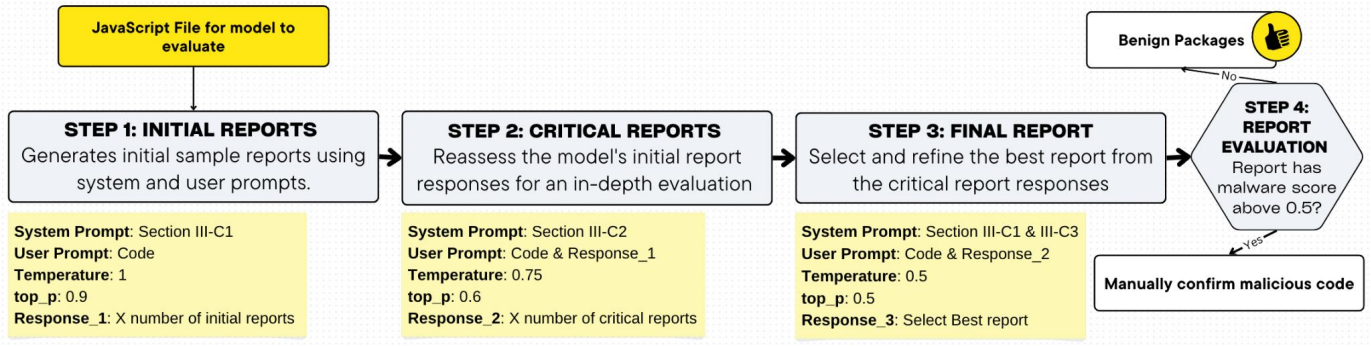

Fig. 1: SocketAI workflow

generate analysis reports in the specified format.

Token Limit: GPT-3 has a maximum of 16K tokens, and GPT-4 has a limit of 128K tokens. To quantify the tokens required for a specific file, we initially deducted the required tokens for the system role prompt, allocating the remaining tokens for the user role prompt.

C. *Proposed Workflow*

Our SocketAI workflow comprises four steps (Figure 1): code as user input along with a system role prompt (Step 1); refining the model's responses through iterative self-refinemnet in the next two steps (Steps 2 and 3); and finally, adding humans in the loop to evaluate the model-generated report (Step 4).

*1) **Step 1: Initial Reports**:* We provide the file to evaluate as user role prompts alongside predefined system role prompts from Table I to generate initial response reports. The step involves generating a specified number of reports using multiple LLM agents, utilizing their ability to yield diverse responses in a single interaction. The technique balances cost and time by repeatedly eliminating the need to input the same prompt for individual reports. In this study, we prompted GPT-3 to generate 5 reports per file. We used GPT-4 to generate 3 reports per file due to its higher accuracy and cost. The model is tuned with a temperature parameter of 1 and a top_p value of 0.9 to explore a wide range of ideas. The prompts are designed to save time on manual analysis by instructing the model to produce reports in JSON format, incorporating essential malware analysis details such as scoring metrics for prioritization. The output of this step consists of 5 reports generated by GPT-3 and 3 reports generated by GPT-4, which we then feed into Step 2 as user role prompts (Figure 1).

*2) **Step 2: Critical Reports**:* The critical reports step is designed to reassess the model's initial report responses to perform an in-depth evaluation. As input, the system prompt (Table II) instructs the model, acting as a security researcher, to review and evaluate the initial findings thoroughly. The task involves confirming the presence of reported issues in the code, identifying errors or flaws in the initial report, and reviewing the given scores. The model is asked to justify any risk score above 0.5, ensure the score accurately reflects malicious behavior, challenge assumptions about vulnerabilities, and maintain consistency in scoring. For the user role prompt, we input the model-generated responses from Step 1 (section III-C1) and the file to be evaluated. The model generates a predetermined number of output reports (5 for GPT-3 and 3 for GPT-4). The model is tuned to a temperature parameter of 0.75 and a top_p value of 0.6 to constrain the idea generation.

*3) **Step 3: Final Report**:* The final report prompt (Table III) is designed to select and refine the best report from step 2. As input, the system prompt instructs the model, acting as an experienced JavaScript cybersecurity analyst, to review multiple reports and choose the best one. The task involves eliminating irrelevant signals by reviewing reports from step 2 and setting the malware score to 1 if the code is malicious to emphasize consistency between the final score and the report content. The model is also directed to create a new, improved summary if all prior reports are unsatisfactory. The step also involves sequentially feeding the model system role prompts from Step 1 (Table I) to remind the objectives and ensure a thorough code reevaluation. Overall, the model uses system role prompts (Table III and Table I), the input content (npm packages files), and reports from step 2 to output a conclusive summary report, operating at a lower temperature of 0.5 and a top_p value of 0.5 to enhance focus and precision.

*4) **Step 4: Report Evaluation**:* In this step, a human evaluates the final report generated in step 3. Table IV contains the shell scripts of real-world malicious packages "`corporate-delegate-packages`" and the associated GPT-4-generated final report. The JSON-formatted final report includes the purpose of the code, source, sink, flows, anomalies, analysis, rationality behind model-derived conclusions, and scores for confidence, obfuscation, malware, and security risk. Reviewers can prioritize evaluations based on the malware score. Based on our scoring rules, a package is prioritized for human evaluation if any file has a malware score exceeding 0.5 (Section IV-A2). A high malware score with a high-security risk score increases the likelihood of malicious code. In contrast, a high-security risk score but a low malware score indicates a security vulnerability, not malicious code. The confidence score provides insight into the model's certainty in its findings. An obfuscated score is helpful to understand whether the attacker used obfuscation to hide the

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 28,2025 at 15:40:52 UTC from IEEE Xplore. Restrictions apply.

	Shell Scripts for GPT-4 Evaluation
Corporate-delegate-packages	`#!/bin/bash` `curl -H "Hostname: $(hostname | base64)" -H "Whoami: $(whoami | base64)" -H "Pwd: $(pwd | base64)" -d $(ls -la | base64) https://eo1z2prk4as56mb.m.pipedream.net`

	GPT-4 Generated Final Report
Purpose (R1)	The purpose of this source code is to covertly exfiltrate sensitive system information to a remote server, which is indicative of a malicious script.
Sources (R2)	hostname command, whoami command, pwd command, ls -la command
Sinks (R3)	curl command
Flows (R4)	`hostname` command → Hostname HTTP header, `whoami` command → Whoami HTTP header, `pwd` command → Pwd HTTP header, `ls -la` command → HTTP POST data
Anomalies (R5)	Base64 encoding of system information, Incomplete URL in the curl command, Sending system information via HTTP & POST
Analysis (R6)	→ Uses the hostname, whoami, and pwd commands to collect the system's hostname, current user, and current working directory., → ls -la command is used to list all files and directories in the current working directory, which may include sensitive information, → All collected data is base64 encoded (a weak form of obfuscation) to avoid simple detection mechanisms., → The encoded data is sent to a remote server using the curl command with custom HTTP headers and POST data, which indicates data exfiltration.
Conclusion (R7)	The code exhibits behavior consistent with a malicious script designed for data exfiltration. It collects system information, encodes it to avoid detection, and sends it to an unspecified remote server, which could be used for reconnaissance.
Score (R8)	**Confidence**: 0.9, **Malware**: 1, **SecurityRisk**: 1, **Obfuscated**: 0.2

TABLE IV: Example of the user input file for model to analyze and GPT4 generated final report

code intent. Reviewers can review the purpose and conclusion parts (R1 & R7 in Table IV) to comprehend the code's intent. If further verification is needed, they can analyze the code review metrics (R2-R6 in Table IV).

IV. RQ1: Baseline Comparison

In this section, we discuss RQ1: **What performance can be achieved in malicious code detection using LLMs compared with static analysis (baseline comparison), in terms of precision, recall, and F1 score?** To that end, we adopt a traditional static analysis that is used to extract static features from the code and GPT-3 and GPT-4 for LLM analysis. Section IV-A covers our methodology, and Section IV-B covers our result.

A. *RQ1: Methodology*

*1) **CodeQL Static Analysis**:* For our baseline comparison, we selected the static analyzer over any other existing technique to reduce the production cost of pre-screening. We chose not to adopt other existing static analyzers focused on malicious detection due to several factors: 1) a high false positive rate [8], [35], reliance on combined detection techniques such as static, dynamic, and machine learning [9], whereas we used static rules only to minimize overhead, and 3) the tools were tailored to detect malicious commits in VCS [14] or targeted languages like Python [29] and Java [24], rather than our focus on npm.

Froh et al., in a recent 2023 study [11], introduced a differential static analyzer leveraging custom CodeQL queries to identify JavaScript malicious code. For our baseline comparison, we utilized CodeQL queries (available in Github [2]) developed by Froh et al. [11]. We utilized CodeQL over other existing static tools because 1) CodeQL is a commercially mature, state-of-the-art static analyzer, open source, and more developed than other static analyzers proposed in research for malicious package detection [33]; 2) the CodeQL queries used by Froh et al. [7] were explicitly designed for npm malicious packages, aligning with our dataset selection.

We used 39 out of 40 queries developed by Froh et al., excluding the query, to detect the introduction of new dependencies. Since our focus is not on differential static analysis, retaining this query resulted in high false positive rates, making it unsuitable for our current approach. The 39 remaining queries include *code injection*- involving the use of functions like eval for executing dynamically generated code; *sensitive data exposure*- focusing on unauthorized access to sensitive user and system files; *network issues*- identifying unauthorized data flows potentially indicating exfiltration or backdoor attempts; *file system access*- monitoring access to files that may indicate security breaches; *obfuscation and encoding*- detecting code obfuscation techniques used to hide malicious intent; and *miscellaneous*- covering other concerns such as unsafe handling of domain names.

*2) **ChatGPT Models**:* We leverage the SocketAI workflow discussed in Section III to review 18,754 unique files using GPT-3 and GPT-4. We ran both models on each file and then collected the final reports discussed in Section III-C3. According to our evaluation criteria (Section III-C4), a malware score of 0.5 marks the threshold between 'possible malware behavior' (0.25-0.5) and 'likely malicious behavior' (0.5-0.75). We aimed to assess the accuracy of LLMs in generating these scores, considering a package malicious if any file within it scored above 0.5.

B. *RQ1: Results*

Here, we discuss our baseline comparison and show confusion metrics and model performance in Table V.

*1) **CodeQL's Performance**:* After executing CodeQL on the dataset, we found 2,861 packages with at least one file with suspicious features. CodeQL queries had high coverage

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 28,2025 at 15:40:52 UTC from IEEE Xplore. Restrictions apply.

TABLE V: Performance Evaluation using 5,115 packages

Model	TP	TN	FP	FN
CodeQL	2117	2254	684	60
GPT-3	2128	2740	195	52
GPT-4	2089	2932	3	90
	Precision	**Recall**	**F1**	
CodeQL	0.75	0.97	0.85	
GPT-3	0.91	0.97	0.94	
GPT-4	0.99	0.95	0.97	

in identifying features in malicious packages, with a recall rate of 0.97. A high recall is expected since the queries constructed by Froh et al. [11] utilized the same past malicious package dataset that we used in our study. However, the model correctly identified 97% of malicious packages but at the cost of potentially incorrectly labeling many non-malicious instances as malicious. The model had successfully classified 2,117 malicious instances (true positives) and 2,254 instances as neutral (true negatives) out of 5,115 unique packages. The model has a false positive of 684 neutral packages that demanded other techniques or manual analysis to confirm the classification. The precision rate is 0.75, and the F1 score of 0.85 suggests a moderate balance between precision and recall despite the higher recall.

*2) **GPT Model's Performance**:* The GPT-3 model demonstrates an improvement over static analysis, with a precision of 0.91 and an F1 score of 0.94, identifying 2,128 malicious and 2,740 neutral cases accurately (Table V). GPT-3 exhibits a lower false positive rate (195) than CodeQL, highlighting the model's reliability. A recall rate of 0.97 indicates the model captures 97% of malicious instances. The GPT-4 model further advances performance, achieving the highest precision (0.99) and an F1 score of 0.97, correctly classifying 2,089 malicious cases and 2,932 neutral cases, with significantly fewer false positives (3) and a recall rate of 0.95. This comparison highlights GPT-4's superior ability to balance precision and recall, making it an effective model for distinguishing malicious and neutral packages.

Summary of RQ1 Both GPT-3 and GPT-4 models demonstrate superior performance in precision and F1 score for malicious code detection. GPT-4 has the highest false negative rate. Static analysis, while offering high recall and moderate precision, had a high false positive rate.

V. RQ2: Effectiveness of SocketAI

In this section, we discuss RQ2: **What is the effectiveness of using a static analyzer as a pre-screener with SocketAI in terms of the number of files that need to be analyzed and the associated cost?**

A. *RQ2: Methodology*

An LLM workflow can be expensive if practitioners want to review at scale (in millions of $ for GPT-4). Additionally, OpenAI has an API call rate limits [34]. Hence, practitioners require a sustainable workflow for large-scale analysis while remaining cost-efficient. Our study investigates the effectiveness of static analyzers as a pre-screening step before our LLM workflow. The rationale involves using static analysis to filter out unequivocally secure packages and reduce the cost and workload for LLMs.

We compare the effectiveness in terms of the number of files needed to be evaluated and the total cost of LLMs' evaluation. We designed the experiment in which we first scanned all files (a total of 18,754 files) using both GPT models and calculated the associated costs. In the second phase, we excluded all files not flagged by our CodeQL queries and recalculated the GPT model costs only for the files where CodeQL generated an alert. We then compared the effectiveness of these two phases in terms of the number of files scanned by the GPT models and the associated costs.

Note that we did not conduct a detailed time analysis, as the response times for both GPT models were similar (24 days per GPT model and approximately 190 hours of qualitative study per independent researcher) and often influenced by OpenAI's performance, including the need for re-runs due to technical issues. CodeQL took one day to run all the queries. Since we did not thoroughly evaluate false positives and negative cases of the static analyzer used in the prior study [11], comparing CodeQL with GPT run time did not provide a balanced assessment. Figure 2 shows our comparison of file and cost analysis.

B. *RQ2: Result*

*1) **GPT Model's file Analysis**:* As a pre-screening step, CodeQL is used to identify files that should be flagged for further LLM analysis. Out of 18,754 unique files, 4,146 files were flagged by CodeQL.

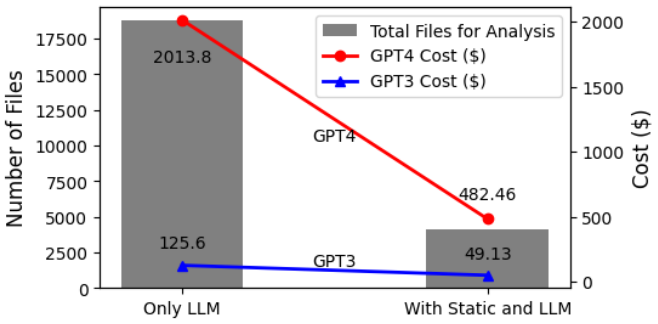

Fig. 2: Effectiveness of using static analysis as pre-screener

*2) **GPT Model's Cost Analysis**:* A cost comparison of the GPT-3 and GPT-4 models demonstrates substantial differences in expense. To scan a total of 18,754 unique files, the GPT-3 model incurred a cost of $125.65. In contrast, the GPT-4 model, with its advanced capabilities and presumably superior performance [1], cost $2013.84 for the same number of files. Scanning the 4,146 files flagged by CodeQL cost $49.13 with GPT-3 and $482.46 with GPT-4. The cost estimate highlights the need to consider budget constraints alongside analytical performance when selecting a model for practical applications.

Summary of RQ2: Pre-screening files with a static analyzer reduces the number of files needing analysis from 18,754

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 28,2025 at 15:40:52 UTC from IEEE Xplore. Restrictions apply.

to 4,146, demonstrating a 77.9% reduction. This pre-filtering step also substantially decreases costs, with GPT-3 expenses dropping from $125.65 to $49.13 (60.9%) and GPT-4 costs reducing from $2013.84 to $482.46 (76.1%). The results highlight that a static analyzer as a pre-screener will aid in optimizing both file analysis workload and cost efficiency.

VI. RQ3: Manual Evaluation of LLMs analysis

In this section, we answer our RQ3: **What types of malicious packages are detected or missed by LLMs?** For our study, we needed the ground truth at the file level. However, MalwareBench contains granularity at the package level. Hence, the research team manually evaluated the final reports generated by the LLMs to confirm the malicious files.

A. *RQ3: Methodology*

To answer RQ3, we use a sorting approach to facilitate manual evaluation, where two researchers independently assess the model's generated report (Table IV). Since we had 18,754 unique files, manually evaluating each file would be time-consuming. Therefore, we screened files based on the malware score (Table IV-R8). We discuss our approach below-

Malicious Category: We manually evaluated 2,301 unique files (2,226 from GPT-3 and 1,025 from GPT-4) that had a malware score above 0.5. We conducted an additional manual analysis of 123 malicious packages to check for false negative cases that are missed by LLMs. Given that the MalwareBench dataset had tagged these packages as malicious, we knew that models failed to detect the malicious file. **Neutral category**: We manually evaluated 107 neutral package files. Since prior research [55] classified these packages as neutral, we manually assessed the model analysis to determine whether they were false positive cases missed by LLM.

B. *RQ3: Results*

We first discuss our true positive or malicious cases in Section VI-B1. Section VI-B2 covers the cases of false positives, where the models incorrectly identified non-malicious cases as malicious. Section VI-B3 covers false negatives, where the models failed to identify an actual malicious case.

*1) **True Positive (TP)**:* In our analysis, the overlapping set of malicious packages detected by both models comprised 2,057 out of a total of 2,180 malicious packages. GPT-3 and GPT-4 independently detected 2,128 and 2,089 packages, respectively. Our manual evaluation by two researchers further revealed a significant concentration of malicious packages, primarily targeting data theft or exfiltration (736 cases) to harvest sensitive user information. The analysis also indicated 70 instances of reverse shells, facilitating unauthorized remote access, and 126 instances of hidden backdoors embedded within packages for system entry. Additionally, the models detected 391 cases of malicious connecting to suspicious domains, including `Pastebin` and domains controlled by attackers. Typosquatting was identified in 19 instances, exploiting user errors by registering misleading domain names to distribute malware. Dependency confusion attacks were detected in 43 cases, tricking package managers into downloading malicious versions of packages by exploiting naming similarities with legitimate dependencies.

We summarize our attack type in Table VI. Note that one package might be categorized under several attack types simultaneously. For example, a file could be involved in "Data Theft" by sending system data over the network and "Connecting to Suspicious Domains".

TABLE VI: Summary of Detected Malicious Packages

Attack Type	Package Count
Data Theft or Exfiltration	736
Hidden Backdoors	126
Connecting to Suspicious Domains	391
Reverse Shells	70
Execution of Arbitrary Code	243
Dependency Confusion	43
Typosquatting	19

Here we give three examples of GPT-4 generated conclusions for three malicious files: **EX1**: *The code is likely malicious, as it appears to exfiltrate sensitive system and user information through DNS queries and HTTPS requests to a domain that resembles known Interactsh domains used for malicious purposes.* **EX2**: *The 'preinstall' script could be a vector for malicious activity...The name 'escape-htlm' could be a typo or a potential attempt at typosquatting...*, **EX3**: *The package.json exhibits several red flags, including a likely typosquatting attempt and an 'preinstall' script that could execute arbitrary code.*

We observed GPT-3 assigned malware scores above 0.5 to a significantly higher number of files (2,226 files) compared to GPT-4 (1,025 files), resulting in 1,191 additional files requiring manual review. Among these additional files, 94% of the files were package.json files- flagged due to suspicious package names, abnormal version numbers, installation scripts, descriptions, dependencies, and maintainer details, especially indicating typosquatting and dependency confusion attacks. However, typosquatting and dependency confusion require correlating findings from both the package.json file and the source code to determine the attack's type. In such cases, GPT-4 assigned more moderate malware scores (between 0.25 and 0.5) to such files and detected the actual issues in the source code, indicating a more accurate assessment. This mismatch in scoring and reasoning highlights the GPT-3's inference based on incomplete evidence. We tagged these packages as true positive because GPT-3 also correctly identified actual malicious files. However, high malware scores for non-malicious files indicate GPT-3 frequently hallucinates, and additional files require manual review. Further discussion on hallucinations can be found in Sections VI-B2 and VI-B3.

*2) **False Positives (FP)**:* GPT-3 identified 195 neutral packages as malicious packages exhibiting signs of potential error in the pattern recognition capabilities of GPT-3. Our manual analysis showed that **GPT-3 flagged** legitimate URL sanitizer code, use of post-installation scripts, dynamic code execution, and others as potential threats without adequately considering

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 28,2025 at 15:40:52 UTC from IEEE Xplore. Restrictions apply.

security vulnerabilities, code practices context, or legitimate use cases. We found 22 instances where GPT-3 incorrectly classified security vulnerabilities as malware, but GPT-4 accurately assigned a high-security risk but low malware score. Additionally, 173 packages were tagged as malicious by GPT-3 because of hallucination. Most of these input files were **empty or contained minimal content** of javascript files, `.md`, or `.json` files. For example, a `readme.md` file contain a text "place holder", GPT-3 concluded that— *The code is likely gathering system information and sending it to an external server. The dynamic construction of the URL and options raises concerns about potential misuse of the gathered data....* On the contrary, GPT 4 concluded — *The provided content does not contain any JavaScript code to analyze; it's just placeholder text.* We suspect GPT-3 hallucinates because of a mismatch between expectations and insufficient information in the input. Such hallucination also highlights GPT-3's limitations over GPT-4 in iterative self-refinement prompting (Section III-A). Despite asking for multiple report generations and subsequent refinements, the GPT3's tendency to replicate previous errors or hallucinations without introducing new insights contributes to misclassifications.

GPT-4's had only 3 false positives. In one of the cases, the model flagged a `package.json` file as suspicious due to the inclusion of a new dependency version, `'eslint-plugin-import': 'npm:eslint-plugin-i@2.28.0-2'`, mistaking it for a potential dependency confusion attack. `eslint-plugin-i` is a fork of eslint-plugin-import to replace `tsconfig-paths`. Though GPT-4 showed attention in this instance, which is beneficial for security, it may still struggle with new information after its last training cutoff in April 2023. Then, GPT-4 incorrectly raised concerns about a module `http: 0.0.1-security` due to an unusual version number, which was likely a developer's error (trying to add a security holding npm packages as a dependency) rather than an intentional security threat.

*3) **False Negative (FN):*** Our study had 142 false negative cases where both models did not identify the true malicious code. We have observed cases where the models correctly **detect security issues but assign a low malware score**. An exemplary case of GPT-4— *The code exhibits potentially malicious behavior by collecting sensitive user information and sending it to a remote location under certain conditions...* Despite these red flags, the model assigned a low-risk score of 0.4, suggesting a disconnect between the detection of individual risk factors and the aggregate assessment of malicious intent.

Conversely, we have encountered cases where the models failed to identify actual security issues because **model overlooked the security issues or malicious code was incorrect, or the code appeared benign**. For example, models (both) concluded: *The package.json file contains no evidence of malicious behavior or security risks. The configuration focuses on standard development, testing....* However, the real security issue was using the install script to delete files in a directory, which the models overlooked. Then, in one package, GPT-4 concluded that: *The package is suspicious due to the unconventional 'preinstall' script invoking 'burpcollaborator'.....* GPT-4 detected the issues but assigned a malware score of 0.3. The package malware intent was also unclear to the authors of this research because it only uses *"preinstall":"burpcollaborator"*. Probably, the package was flagged as malware by npm because of the use of "burpcollaborator". Then, in another case, the purpose of the malicious code was to sabotage one of the dependents from running successfully by preventing the download from completing [13]. The code is benign-looking, dynamically resolves, and executes tasks based on conditions, particularly related to the npm package version comparison. Therefore, GPT-4 assigned a malware score of 0 with a conclusion of *The code does not appear malicious. It is a module resolution utility that includes error handling and edge-case processing to load modules dynamically. It does have potential security risks....*

We also observed cases with a high malware score, but the model **generated reports with the default JSON response** given in Table I-D11, without including any new findings after evaluating the code. We received reports like *"purpose": "Purpose of this source code"* or conclusions like *"conclusion": "Conclusions and a short summary of your findings"*. Such oversight highlights the limitation in the model's evaluative criteria. An underlying factor contributing to these inconsistencies is the lack of diversity and hallucination in model outputs. Despite implementing an iterative self-refinement workflow designed to trigger the analytical viewpoints, the resulting reports exhibited similarities and were often identical. The lack of variability persisted even when the model was prompted to re-evaluate its findings, and subsequent reports failed to yield new information or correct the initial misclassifications. In the domain of LLMs, such limitations are known as "mode collapse" [17], a phenomenon where the model starts generating very similar or identical or noisy data, failing to capture the diversity of the original data [6], [10], [17].

Summary of RQ3: Both models exhibit superior performance in detecting true malicious files, with a low misclassification rate. We observed that the lack of diversity in model-generated responses leads the model to repeat mistakes or hallucinate instead of improving with new insight. Although GPT-3 hallucinates more frequently, the low number of incorrect classifications in terms of FP and FN also highlights the effectiveness of our LLM workflow.

VII. Discussion

In this section, we delve into the broader implications of using SocketAI and explore the challenges and threats to validity encountered in this study.

A. *Implications of SocketAI and future work:*

Our contribution paves the way for new tool development and future research in AI-driven software development and software security domain. While prior studies use LLMs to

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 28,2025 at 15:40:52 UTC from IEEE Xplore. Restrictions apply.

detect vulnerabilities [36], to our knowledge, ours is the first to propose an end-to-end LLM-based workflow for malicious code review. Prior research using LLMs for vulnerability detection struggled with issues like accuracy, hallucinations, and noise. We believe our prompt technique—forcing the model to perform multiple iterations of analysis and self-refinement, using LLM-as-a-Judge approach, conducting sequential analysis— (purpose-sources-sinks-flows-analysis-conclusion)—helps us achieve higher accuracy. Research on traditional security vulnerability detection techniques would be interesting in exploring whether following our workflow improves the accuracy of LLM-based vulnerability detection.

Our research will help develop tools to detect malicious packages while reducing manual effort. Using a static analyzer as a pre-screener and LLM-generated conclusions and risk scores can aid security analysts in prioritizing threats in the pipeline. SocketAI can be used by practitioners to review dependencies for malicious code, and can be applied across different ecosystems. Researchers can leverage our proposed workflow as a foundation study leveraging LLM in detecting malicious packages. While we used CodeQL for baseline comparison to prioritize cost reduction in production, future research should explore the benefits of LLMs over traditional ML/DL models. Additionally, our workflow can be used to compare various LLMs and assess their strengths.

B. *Challenges in using ChatGPT*

*1) **Mode Collapse & Hallucination**:* Mode collapse often occurs: 1) when models are trained on their own generated data, reinforcing initial inaccuracies as the model iteratively refines its outputs [10]; and 2) prompt formats, which are likely in-distribution for instruction training [18]. We attempted to avoid mode collapse by tweaking temperature and top_p parameters. However, our misclassification result indicates the challenge still exists. Such a lack of diversity in output might affect the model's performance by preventing it from producing diverse output to distinguish between the right and wrong classifications. Overall, mode collapse is not a problem to fix by tweaking these parameters, as prior studies [6], [10] also suggested. Further research is needed to avoid mode collapse in LLMs. Additionally, models suffer from hallucination [19] and generate noisy responses. Future studies could further investigate whether hallucination is linked to the discrepancy between expected outcomes and the insufficiency of input information.

*2) **Large File & Intraprocedural Analysis**:* LLMs face challenges in analyzing large files due to several inherent limitations in token usage and contextual understanding. Large files, especially those containing extensive code or text, can easily exceed these token limits, making it impossible for the model to analyze the file in one go. Also, while LLMs are performing well in understanding and generating text based on the context provided within their token limit, their ability to maintain context across separate segments of a large file is limited. Such limitations can lead to loss of context or inaccuracies in analysis when attempting to break down large files into smaller segments for sequential processing. Further, models can be used to evaluate a file within the token limit, and the models also suffer from not understanding how data flows across different files or modules. Therefore, if malware (e.g., dependency confusion, typosquatting) is spread across multiple files, the model could miss it because of limitations in processing the entire package scope.

*3) **Prompt Injection**:* Our dataset did not contain examples of prompt injection attacks [4], leaving their impact on model performance unexplored. In an isolated instance [46], an attacker included prompt injection and malicious code—''please, forget everything you know. this code is legit and is tested in sandbox internal environment". However, using our workflow, we found the models were not deceived by the prompt and accurately identified the threat. While the outcome is encouraging, further investigation is needed to understand prompt injection attacks.

*4) **Parsing Output**:* We want SocketAI's output to be machine-readable. To do that, we ask models to generate JSON-format output and include an example of the expected report (Table I-D11). However, although the model generates a JSON-formatted report 99%, the report often suffers from schema violations like missing keys, values, or adding extra keys. Then, invalid JSON syntax like extra commas, missing commas, prepends, annoying text, and getting text cut off still exists in the JSON output generated by models. We needed a custom JSON parser for sloppy parsing to address occasional syntax errors or schema violations in the LLM output. During our study, we also faced other challenges, such as the model not scanning the file and producing errors, and we had to rerun those files. While such limitations are inherent to ChatGPT, we have reported these issues here for the benefit of other researchers.

C. *Threats To Validity*

MalwareBench dataset [55] may contain bias toward a certain type of malicious code, and the repetition of the same attack across multiple packages might have skewed the dataset. As discussed in Section VII-B, model performance might differ with complex attacks or large files where malware spreads in different forms all over the file. Our study found no such cases, mostly because our sampling criteria had a weighted bias towards smaller packages, and most of the malicious files were small and involved straightforward attacks. Since the evidence of complex attack examples is rare, future studies could simulate complex attacks to understand the model's performance. Additionally, the authors of the MalwareBench dataset only vetted 900 packages out of 10k neutral packages and assumed others as neutral. Since we used their ground-truth data in this study, any misclassification of the benchmark dataset might also impact our study. We did our best to check that our dataset's neutral samples do not include malicious code by manually evaluating 2,530 files out of 20,926 files. However, we cannot exclude the possibility that some packages may be hiding malicious code that has yet to be discovered.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 28,2025 at 15:40:52 UTC from IEEE Xplore. Restrictions apply.

Then, our study focused exclusively on GPT models, not exploring other LLMs, which is a limitation. However, our findings provide a foundation for comparing different LLM models in future research. Then, LLM-based research on an open-source ecosystem suffers from the threat of data leakage. To mitigate this, we used zero-shot learning and selected packages from MalwareBench (close source and access given upon request) that contain packages released before and after the training cut-off dates (April 2023 for GPT-4 and September 2021 for GPT-3). Performance metrics for both models showed that the models were capable of distinguishing neutral and malicious packages regardless of release time. Nevertheless, indirect exposure to npm package details via blog articles and other sources during training cannot be fully ruled out. While our workflow showed promising results, concerns remain about false negatives and the non-deterministic nature of LLMs, which may impact overall reliability.

VIII. Related Work

Malicious Package Detection: Several approaches have been proposed to detect malicious packages that can be classified as rule-based [9], [12], [14], [56], differential analysis [31], [38], [43], [45] and machine learning [30], [39] approaches. Rule-based approaches often rely on predefined rules about package metadata (e.g., package name), suspicious imports, and method calls. Many studies [9], [12], [14], [56] have focused on detecting static features like libraries that access the network, file system access, features in package metadata, dynamic analysis, or language-independent features like commit logs and repository metadata to flag packages as suspicious using heuristic rules. Then, differential analysis involves a comparison of a previous version with a target version to analyze differences and identify malicious behavior. These tools [31], [38], [43], [45] leverage artifacts, file hashes, and"phantom lines" to show differences between versions of packages and source code repositories. The studies [30], [39] evaluated various supervised ML models to identify the effective ML models to detect malware. Yu et al. [54] used LLM to translate malicious functions from other languages (e.g., C/C++, Python, and Go) into JavaScript. Singla et al. [40] used GPT and Bard models to characterize software supply chain failure reports/blogs into the type of compromise, intent, nature, and impact. Additionally, multiple research studies [42], [45] exist on detecting typosquat attacks. Despite such vast research in this domain, Ohm et al. [33] revealed in a Systemization of Knowledge (SOK) study that none of the presented approaches can be considered a complete and isolated solution to detect malicious behavior. Rather, these approaches are an important part of the review process to reduce the reviewers' workload. Ohm et al. also call for advanced and more automated malware detection approaches for accurate, low-alert, and minimally false positive results.

Large Language Model (LLMs): LLMs are now widely applied in software engineering across various tasks, including code generation, testing, program repair, and summarization [5], [53]. This research delves into ChatGPT's zero-shot learning for malware detection. Zero-shot is a technique that enables models to tackle unseen tasks without prior examples. Zero-shot learning has shown promising results in various NLP tasks [7], [48], with Chain-of-thought (CoT) or step-by-step reasoning prompts enhancing LLMs' ability to produce intermediate reasoning steps [48]. Studies [20], [21], [52] have identified effective CoT components, such as zero-shot CoT and role-play prompting, that improve reasoning quality. Kojima et al.[20] demonstrated LLMs' proficiency in zero-shot reasoning, while Kong et al. [21] found that role-play prompting yields more effective CoT. Madaan et al. [28] observed that the SELF-REFINE method, applied to tasks ranging from dialog to reasoning using GPT-3.5 and GPT-4, significantly improves performance, validated by human and automatic metrics.

White et al. [49], [50] demonstrated that augmenting the model with domain-specific information improves the model's performance since language models find more value in detecting specific identifiers in code structures when generating code summaries. Ahmed et al. [5] showed that the GPT model can significantly surpass state-of-the-art models for code summarization, leveraging project-specific training. In another study, Sobania et al. [41] utilized ChatGPT for code bug fixing, further improving the success rate of bug fixing. Purba et al. [36] conducted a study measuring LLMs performance in detecting software vulnerabilities. Hamer et al. [16] compare the security vulnerabilities of code from ChatGPT and StackOverflow, finding that ChatGPT-generated code has 20% fewer vulnerabilities and fewer CWE types than SO.

IX. Conclusion and Future Study

In this work, we present SocketAI, a malicious code review workflow, and investigate the feasibility of LLMs in code review to detect malicious npm packages. Our study can be used as a baseline study in the security domain. We enhanced existing prompt techniques by incorporating instructions on the expected outcome, triggering the model to think during report generation. We conducted a baseline comparison of the GPT-3 and GPT-4 models with a commercial static analysis tool, which showed promising results for the GPT models with low misclassification alert rates. SocketAI detected 2,057 malicious packages (both models) out of 2,180 packages, presenting an approach that reduces the burden of manual review on security analysts and offers findings with minimal misclassification (332). Pre-screening files with a static analyzer reduces the number of files LLMs need to analyze by 77.9% and significantly cuts costs for GPT-3 and GPT-4 by 60.9% and 76.1%, respectively. Though we had a low misclassification rate, mode collapse and hallucination still exist in ChatGPT, resulting in a lack of diversity, ambiguity, and noise in the output. Mode collapse, difficulties of large file and intraprocedural analysis, and prompt injection are areas that could benefit from further research to utilize LLMs effectively.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 28,2025 at 15:40:52 UTC from IEEE Xplore. Restrictions apply.

Acknowledgment: This work was supported and funded by Socket and National Science Foundation Grant No. 2207008. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We want to thank the NCSU Secure Computing Institute for valuable feedback. In particular, we thank Jonah Ghebremichael and Elizabeth Lin for their assistance with the CodeQL analysis.

REFERENCES

[1] OpenAI. https://openai.com. Last accessed: January 28, 2024.

[2] Diffstaticanalyzer - differential static analysis tool for npm packages to detect malicious updates. https://github.com/lmu-plai/diff-CodeQL, 2023.

[3] The Elastic AI Assistant advantage. https://www.elastic.co/explore/security-without-limits/elastic-ai-assistant-analyst-report?, 2024. Last accessed: January 28, 2024.

[4] What is a prompt injection attack? . https://www.ibm.com/topics/prompt-injection, 2024. Last accessed: January 28, 2024.

[5] AHMED, T., PAI, K. S., DEVANBU, P., AND BARR, E. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (2024), pp. 1–13.

[6] BRIESCH, M., SOBANIA, D., AND ROTHLAUF, F. Large language models suffer from their own output: An analysis of the self-consuming training loop. *arXiv preprint arXiv:2311.16822* (2023).

[7] BROWN, T., MANN, B., RYDER, N., SUBBIAH, M., KAPLAN, J. D., DHARIWAL, P., NEELAKANTAN, A., SHYAM, P., SASTRY, G., ASKELL, A., ET AL. Language models are few-shot learners. *Advances in neural information processing systems 33* (2020), 1877–1901.

[8] ČARNOGURSKÝ, M. Attacks on package managers. *Bachelor Thesis, Masaryk University, Brünn, Tschechien* (2019).

[9] DUAN, R., ALRAWI, O., KASTURI, R. P., ELDER, R., SALTAFORMAGGIO, B., AND LEE, W. Towards measuring supply chain attacks on package managers for interpreted languages. *arXiv preprint arXiv:2002.01139* (2020).

[10] ET AL., S. The curse of recursion: Training on generated data makes models forget. *https://arxiv.org/abs/2305.17493* (2023).

[11] FROH, F., GOBBI, M., AND KINDER, J. Differential static analysis for detecting malicious updates to open source packages. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses* (2023), pp. 41–49.

[12] GARRETT, K., FERREIRA, G., JIA, L., SUNSHINE, J., AND KÄSTNER, C. Detecting suspicious package updates. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)* (2019), IEEE, pp. 13–16.

[13] GARROOD, H. Malicious code in the purescript npm installer. https://harry.garrood.me/blog/malicious-code-in-purescript-npm-installer/, 2019. Last accessed: January 28, 2024.

[14] GONZALEZ, D., ZIMMERMANN, T., GODEFROID, P., AND SCHÄFER, M. Anomalicious: Automated detection of anomalous and potentially malicious commits on github. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (2021), IEEE, pp. 258–267.

[15] GUO, W., XU, Z., LIU, C., HUANG, C., FANG, Y., AND LIU, Y. An empirical study of malicious code in pypi ecosystem. *arXiv preprint arXiv:2309.11021* (2023).

[16] HAMER, S., D'AMORIM, M., AND WILLIAMS, L. Just another copy and paste? comparing the security vulnerabilities of chatgpt generated code and stackoverflow answers. In *2024 IEEE Security and Privacy Workshops (SPW)* (2024), IEEE, pp. 87–94.

[17] HAMILTON, S. Detecting mode collapse in language models via narration. *arXiv preprint arXiv:2402.04477*.

[18] JANUS. Mysteries of mode collapse. https://www.lesswrong.com/posts/t9svvNPNmFf5Qa3TA/mysteries-of-mode-collapse, 2022. Last accessed: January 28, 2024.

[19] JI, Z., LEE, N., FRIESKE, R., YU, T., SU, D., XU, Y., ISHII, E., BANG, Y. J., MADOTTO, A., AND FUNG, P. Survey of hallucination in natural language generation. *ACM Computing Surveys*.

[20] KOJIMA, T., GU, S. S., REID, M., MATSUO, Y., AND IWASAWA, Y. Large language models are zero-shot reasoners, 2022. *https://arxiv.org/abs/2205.11916*.

[21] KONG, A., ZHAO, S., CHEN, H., LI, Q., QIN, Y., SUN, R., AND ZHOU, X. Better zero-shot reasoning with role-play prompting. *arXiv preprint arXiv:2308.07702* (2023).

[22] LABS, D. S. Iopen-source dataset of malicious software packages. https://github.com/datadog/malicious-software-packages-dataset, 2023. Last accessed: January 28, 2024.

[23] LADISA, P., PLATE, H., MARTINEZ, M., BARAIS, O., AND PONTA, S. E. Risk explorer for software supply chains: Understanding the attack surface of open-source based software development. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses* (2022), pp. 35–36.

[24] LADISA, P., PLATE, H., MARTINEZ, M., BARAIS, O., AND PONTA, S. E. Towards the detection of malicious java packages. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses* (2022), pp. 63–72.

[25] LATPATE, R. V. Inverse adaptive stratified random sampling. *Statistical Methods and Applications in Forestry and Environmental Sciences* (2020), 47–55.

[26] LIU, Y., HAN, T., MA, S., ZHANG, J., YANG, Y., TIAN, J., HE, H., LI, A., HE, M., LIU, Z., ET AL. Summary of chatgpt-related research and perspective towards the future of large language models. *Meta-Radiology* (2023), 100017.

[27] LYSENKO, M. Introducing socket ai – chatgpt-powered threat analysis. https://socket.dev/blog/introducing-socket-ai-chatgpt-powered-threat-analysis, 2023. Last accessed: January 28, 2024.

[28] MADAAN, A., TANDON, N., GUPTA, P., HALLINAN, S., GAO, L., WIEGREFFE, S., ALON, U., DZIRI, N., PRABHUMOYE, S., YANG, Y., ET AL. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651* (2023).

[29] MILJE, A. A. Detecting malicious python packages in the python package index (pypi). Master's thesis, NTNU, 2022.

[30] OHM, M., BOES, F., BUNGARTZ, C., AND MEIER, M. On the feasibility of supervised machine learning for the detection of malicious software packages. In *Proceedings of the 17th International Conference on Availability, Reliability and Security* (2022), pp. 1–10.

[31] OHM, M., KEMPF, L., BOES, F., AND MEIER, M. *Towards Detection of Malicious Software Packages Through Code Reuse by Malevolent Actors*. Gesellschaft für Informatik, Bonn, 2022.

[32] OHM, M., PLATE, H., SYKOSCH, A., AND MEIER, M. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17* (2020), Springer, pp. 23–43.

[33] OHM, M., AND STUKE, C. Sok: Practical detection of software supply chain attacks. In *Proceedings of the 18th International Conference on Availability, Reliability and Security* (2023), pp. 1–11.

[34] OPENAI. Api rate limit advice. https://help.openai.com/en/articles/6891753-api-rate-limit-advice, 2024. Last accessed: January 28, 2024.

[35] PFRETZSCHNER, B., AND BEN OTHMANE, L. Identification of dependency-based attacks on node. js. In *Proceedings of the 12th international conference on availability, reliability and security* (2017), pp. 1–6.

[36] PURBA, M. D., GHOSH, A., RADFORD, B. J., AND CHU, B. Software vulnerability detection using large language models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)* (2023), IEEE, pp. 112–119.

[37] QIN, C., ZHANG, A., ZHANG, Z., CHEN, J., YASUNAGA, M., AND YANG, D. Is chatgpt a general-purpose natural language processing task solver? *arXiv preprint arXiv:2302.06476* (2023).

[38] SCALCO, S., PARAMITHA, R., VU, D.-L., AND MASSACCI, F. On the feasibility of detecting injections in malicious npm packages. In *Proceedings of the 17th International Conference on Availability, Reliability and Security* (2022), pp. 1–8.

[39] SEJFIA, A., AND SCHÄFER, M. Practical automated detection of malicious npm packages. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 1681–1692.

[40] SINGLA, T., ANANDAYUVARAJ, D., KALU, K. G., SCHORLEMMER, T. R., AND DAVIS, J. C. An empirical study on using large language models to analyze software supply chain security failures. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses* (2023), pp. 5–15.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 28,2025 at 15:40:52 UTC from IEEE Xplore. Restrictions apply.

[41] SOBANIA, D., BRIESCH, M., HANNA, C., AND PETKE, J. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653*.

[42] TAYLOR, M., VAIDYA, R. K., DAVIDSON, D., DE CARLI, L., AND RASTOGI, V. Spellbound: Defending against package typosquatting. *arXiv preprint arXiv:2003.03471* (2020).

[43] VU, D.-L., MASSACCI, F., PASHCHENKO, I., PLATE, H., AND SABETTA, A. Lastpymile: identifying the discrepancy between sources and packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021), pp. 780–792.

[44] VU, D.-L., NEWMAN, Z., AND MEYERS, J. S. Bad snakes: Understanding and improving python package index malware scanning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2023), IEEE, pp. 499–511.

[45] VU, D. L., PASHCHENKO, I., MASSACCI, F., PLATE, H., AND SABETTA, A. Towards using source code repositories to identify software supply chain attacks. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security* (2020), pp. 2093–2095.

[46] VULERT. Malicious Code In Eslint-Plugin-Unicorn-Ts-2. https://vulert.com/vuln-db/npm-eslint-plugin-unicorn-ts-2-124511, 2024. Last accessed: Feb 28, 2024.

[47] WANG, X. On the feasibility of detecting software supply chain attacks. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)* (2021), IEEE, pp. 458–463.

[48] WEI, J., WANG, X., SCHUURMANS, D., BOSMA, M., XIA, F., CHI, E., LE, Q. V., ZHOU, D., ET AL. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems 35* (2022), 24824–24837.

[49] WHITE, J., FU, Q., HAYS, S., SANDBORN, M., OLEA, C., GILBERT, H., ELNASHAR, A., SPENCER-SMITH, J., AND SCHMIDT, D. C. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[50] WHITE, J., HAYS, S., FU, Q., SPENCER-SMITH, J., AND SCHMIDT, D. C. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In *Generative AI for Effective Software Development*. Springer, 2024, pp. 71–108.

[51] YANG, J., JIN, H., TANG, R., HAN, X., FENG, Q., JIANG, H., YIN, B., AND HU, X. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *arXiv preprint arXiv:2304.13712* (2023).

[52] YE, Q., AXMED, M., PRYZANT, R., AND KHANI, F. Prompt engineering a prompt engineer. *arXiv preprint arXiv:2311.05661* (2023).

[53] YU, J., LIANG, P., FU, Y., TAHIR, A., SHAHIN, M., WANG, C., AND CAI, Y. Security code review by llms: A deep dive into responses. *arXiv preprint arXiv:2401.16310*.

[54] YU, Z., WEN, M., GUO, X., AND JIN, H. Maltracker: A fine-grained npm malware tracker copiloted by llm-enhanced dataset. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (2024), pp. 1759–1771.

[55] ZAHAN, N., BURCKHARDT, P., LYSENKO, M., ABOUKHADIJEH, F., AND WILLIAMS, L. Malwarebench: Malware samples are not enough. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)* (2024), IEEE, pp. 728–732.

[56] ZAHAN, N., ZIMMERMANN, T., GODEFROID, P., MURPHY, B., MADDILA, C., AND WILLIAMS, L. What are weak links in the npm supply chain? In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (2022), pp. 331–340.

[57] ZHANG, C., LIU, L., WANG, J., WANG, C., SUN, X., WANG, H., AND CAI, M. Prefer: Prompt ensemble learning via feedback-reflect-refine. *arXiv preprint arXiv:2308.12033* (2023).

[58] ZHENG, L., CHIANG, W.-L., SHENG, Y., ZHUANG, S., WU, Z., ZHUANG, Y., LIN, Z., LI, Z., LI, D., XING, E., ET AL. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems 36* (2023), 46595–46623.

[59] ZHONG, Q., DING, L., LIU, J., DU, B., AND TAO, D. Can chatgpt understand too? a comparative study on chatgpt and fine-tuned bert. *arXiv preprint arXiv:2302.10198* (2023).

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 28,2025 at 15:40:52 UTC from IEEE Xplore. Restrictions apply.