An Empirical Study on Using Large Language Models to Analyze Software Supply Chain Security Failures

Tanmay Singla∗
Purdue University
West Lafayette, IN, USA
singlat@purdue.edu

Dharun Anandayuvaraj∗
Purdue University
West Lafayette, IN, USA
dananday@purdue.edu

Kelechi G. Kalu
Purdue University
West Lafayette, IN, USA
kalu@purdue.edu

Taylor R. Schorlemmer
Purdue University
West Lafayette, IN, USA
tschorle@purdue.edu

James C. Davis
Purdue University
West Lafayette, IN, USA
davisjam@purdue.edu

ABSTRACT
As we increasingly depend on software systems, the consequences of breaches in the software supply chain become more severe. High-profile cyber attacks like SolarWinds and ShadowHammer have resulted in significant financial and data losses, underlining the need for stronger cybersecurity. One way to prevent future breaches is by studying past failures. However, traditional methods of analyzing past failures require manually reading and summarizing reports about them. Automated support could reduce costs and allow analysis of more failures. Natural Language Processing (NLP) techniques such as Large Language Models (LLMs) could be leveraged to assist the analysis of failures.

In this study, we assessed the ability of Large Language Models (LLMs) to analyze historical software supply chain breaches. We used LLMs to replicate the manual analysis of 69 software supply chain security failures performed by members of the Cloud Native Computing Foundation (CNCF). We developed prompts for LLMs to categorize these by four dimensions: type of compromise, intent, nature, and impact. GPT 3.5’s categorizations had an average accuracy of 68% and Bard’s had an accuracy of 38% over these dimensions. We report that LLMs effectively characterize software supply chain failures when the source articles are detailed enough for consensus among manual analysts, but cannot yet replace human analysts. Future work can improve LLM performance in this context, and study a broader range of articles and failures.

CCS CONCEPTS
• Security and privacy → Software and application security; • General and reference → Empirical studies; • Software and its engineering → Software defect analysis.

KEYWORDS

1 INTRODUCTION
Software mediates almost all aspects of modern life [48]. To reduce development time, software applications integrate dependencies both directly (e.g., importing a library) and indirectly (e.g., that library’s dependencies). These dependencies may come to dominate the application’s risk profile: it has been estimated that the source code of a typical web application is comprised of 80% dependencies and only 20% custom business logic [68, 82]. The owners of these dependencies may be external to the organization developing the application, and thus the reduction of development time comes with an increase in risks associated with this software supply chain [24, 40]. One potential risk is a software supply chain attack — actors insert or exploit vulnerable logic in dependencies, these dependencies are integrated into applications, and the vulnerability becomes exploitable in application deployments [63].

In a failure-aware engineering process, engineers study past failures to prevent future ones [4, 70]. Although organizations may be unwilling to publicly disclose their own failures, news articles and other kinds of grey literature could provide sufficient information on failures [3]. Such data comprises “Open-Source Intelligence” [84], and are used by governmental bodies, military institutions, and law enforcement agencies [39] to design security offenses and defenses.

Current approaches to garnering open-source intelligence, e.g., studying news articles of failures, require costly expert manual analysis. For example, the Cloud Native Computing Foundation (CNCF) maintains a collection of software supply chain security failures analyzed manually — a “Catalog of Supply Chain Compromises” [13]. This catalog has been further analyzed manually [36]. With the goal of reducing the costs of manual analysis, we assess the effectiveness of Large Language Models (LLMs) in gathering open-source intelligence. An overview of the proposed use of LLMs
This paradigm shift leads to software supply chain: the collection of systems, devices, and people which result in a final software product [29]. Figure 2 provides an illustration. According to Google [41], the constituents of a software supply chain include: (1) The code developed by teams, its dependencies, and the various internal and external software applications utilized in the development, compilation, packaging, and installation of the software; (2) The rules and procedures used in all stages of the process; and (3) The systems used for the development of the software and its dependencies. A software supply chain can also be viewed as a network linking actors who perform operations on artifacts [24, 61, 63].

2 BACKGROUND AND RELATED WORK

2.1 Software Supply Chain

Over the years, software production has changed significantly. Early software engineers wrote most code from scratch, increasing production costs [88]. As reusable libraries and frameworks became more available, software engineers shifted to more software reuse [80]. Software applications now commonly rely on external code components, often referred to as dependencies. These dependencies, including packages, libraries, frameworks, and other artifacts, serve as building blocks in modern software development [80].

Figure 1: Proposed use of Large Language Models (LLMs) to analyze software supply chain failures. Failures are often reported in articles and blogs. Organizations concerned with cybersecurity (e.g., governments, corporations) manually analyze failure reports. We evaluate LLMs as an aid.

Figure 2: A Software Ecosystem’s Supply Chain Component and Dependency Vulnerability Flow.

The popularity and reliance on third-party dependencies have been reported in various studies. For example, a 2012 study by Nikiforakis et al. [60] showed that 88% of the Alexa top 10,000 websites included at least one remote JavaScript library. Also, according to a 2019 Synopsys Black Duck report, over 96% of the applications they analyzed include some OSS libraries. These libraries often make up more than 50% of the average code-base [72]. In the 2023 version of this report, the percentage of code in codebases that was open source had risen to about 80% [68, 82].

Software supply chains come with a tradeoff. Costs are reduced during product development and maintenance, but harm may result due to a mismatch between the desired integrity level of a product and the integrity level achieved by one’s dependencies. Defects in dependencies may cause an application to fail, as we discuss next.

2.2 Software Supply Chain Attacks

Faults in software supply chains leave applications vulnerable to attack [85]. Attacks on software supply chains (or records about them) are a recent trend, following the industry shift to relying on third-party components (§2.1). According to a 2021 Sonatype report [80], from February 2015–June 2019 only 216 software supply chain attacks were recorded, then from July 2019 to May 2020 there were 929 attacks recorded, and from 2020-2021 there were over 12,000 attacks recorded. In their 2022 report, this number skyrocketed to 88,000 [81]. Some high-profile attacks, such as SolarWinds [43] and ShadowHammer [49], threatened US national security.

These and similar attacks have inspired comments from many organizations. Governmental organizations such as the Cybersecurity and Infrastructure Security Agency (CISA), the National Security...
Agency (NSA), and the European Union Agency for Cybersecurity (ENISA) have published threat reports and guidance for securing software supply chains [28, 30]. Industry organizations such as the Cloud Native Computing Foundation (CNCF) have also published their own findings and suggestions [78]. These findings have led to the development of security frameworks such as the widely-recognized Supply-chain Levels for Software Artifacts (SLSA) [83].

Academics have also begun to focus on software supply chain attacks. Ohm et al. [62], Ladisa et al. [52], Zimmerman et al. [96], and Zahan et al. [92] studied and characterized attacks on the software supply chain. Okafor et al. [63] condensed existing knowledge about software supply chain attacks into a four-stage attack pattern consisting of initial compromise, alteration, propagation, and exploitation. Table 1 summarizes many avenues for these attacks.

2.3 Failure Studies in Software Engineering
Software engineers have finite resources to produce software [79]. Engineers accept some defects [19, 51], but try to eliminate severe defects that may cause incidents: undesired, unplanned, software-induced events that incur substantial loss [53]. Some defects are vulnerabilities: defects that may be exploited to compromise the security or integrity of a system [9]. Intentional exploitation of vulnerabilities to compromise the security, privacy, or functionality of a system is a malicious attack. Whether severe defects are caught internally or result in incidents, their presence is a failure indicating a flawed software engineering process.

All engineered systems will fail, regardless of the process (e.g., Agile or Plan-based) and methods (e.g., test-driven development or formal methods). For example, Fonseca et al. identified 16 defects across three formally verified systems due to invalid assumptions about the software environment [33]. Across all schools of software engineering thought, from ISO to Agile, guidelines agree that software engineers should analyze failures to improve for next time [7, 8, 16, 31, 32, 38, 45–47, 50]. In light of this, techniques to learn from failures [12] as well as to manage the resulting knowledge [23] are important software engineering knowledge.

Many researchers have studied software failures in an effort to learn from them [1, 3, 20, 36]. This failure analysis research has advanced the software engineering field [3, 53, 59]. However, the high costs associated with failure analysis methods — which rely on manual analysis — deter many organizations from undertaking failure analysis [69]. In their literature review, Amusuo et al. noted that the typical methodology of academic failure analysis is also manual analysis, and recommended the evaluation of Natural Language Processing (NLP) tools to assist in these tasks [2]. Our study responds by evaluating NLP tools in the context of analyzing cybersecurity failures in the software supply chain.

2.4 Natural Language Processing in Support of Software Engineering

2.4.1 NLP to Analyze Supply Chain Failures. In §2.2 we noted that many governments, companies, and academics are studying software supply chain failures. To the best of our knowledge, these studies are conducted manually. This reduces the number of organizations that can gather such intelligence, and we expect that manual efforts will not scale as the number of software supply chain attacks continues to increase.

We believe that recent progress in NLP (Natural Language Processing) could enable large-scale analysis of supply chain failures. Specifically, recent advancements in Large Language Models (LLMs) could aid in studying supply chain failures. LLMs are neural network-based language models that predict the next word based on the most recent context and past words [95]. We therefore hypothesize they could extract relevant failure information from software supply chain failure data sources. While LLMs have been evaluated on many natural language tasks, we are not aware of their prior application to this topic.

2.4.2 Other Applications of NLP in Software Engineering. Natural Language Processing (NLP) has been leveraged for various phases of the Software Development Life-Cycle (SDLC). NLP tools have been proposed for: (1) specification, to detect, extract, model, trace, and classify tasks [94]; (2) design, to model software systems [75], (3) development, to generate code and to detect vulnerabilities and [27]; (4) testing [34]; (5) deployment, to identify risks [87]; and (6) maintenance, to classify user feedback [67]. In this paper, we apply NLP tools to learn from software supply chain failures.

3 RESEARCH QUESTIONS
To reduce the costs of analyzing software supply chain failures, we explore the effectiveness of Large Language Models (LLMs) in automating the analysis of these failures. Towards this goal, we used LLMs to replicate a manual study of software supply chain failures [13]. Specifically, we investigate:

- **RQ1.** How effective are LLMs in replicating manual analysis of software supply chain failures?

- **RQ2.** Do LLMs suggest viable mitigation strategies for preventing future failures?

4 METHODOLOGY
An overview of our methodology is illustrated in Figure 3. To assess the effectiveness of LLMs at replicating manual analysis of software supply chain failures, we compare the analysis of a manually generated catalog against the responses generated by two popular LLMs: ChatGPT [76] and Bard [71]. Specifically to replicate the catalog, we engineered prompts for the LLMs to extract type of compromise, intent, nature, and impact information from the source blogs and news reports. Additionally, we constructed a prompt to gather lessons learned, similar to a postmortem [6]. We evaluate the LLM generated catalog for correctness against the CNCF baseline manual catalog. We manually extract the intent, nature, and impact information and compare against the LLM’s extraction, to evaluate the LLM’s effectiveness at conducting an extended failure analysis.

4.1 Articles for analysis
The CNCF’s “Catalog of Supply Chain Compromises” was used as the baseline dataset [13]. We are not aware of an alternative dataset. This is a catalog of 69 software supply chain security failures analyzed from news articles and blogs from 1984-2022. Each entry
describes the failure and its impacts. Each of the entries was a failure with a link to a news-style article, written in English, describing the failure in further detail. These articles were not authored by the CNCF, but were rather deemed by CNCF to be good descriptions of the failure. The contents of the articles varied in their level of detail. Each entry in CNCF’s catalog had a corresponding type from the options listed in Table 1. Some examples are in Table 2.  

### 4.2 Dimensions of analysis

The dimensions of analysis that we replicate and conduct for the software supply chain failures are outlined in Table 3. Additionally, we extend the analysis of the articles in the catalog to explore the capabilities of LLMs at analyzing failures based on data commonly collected to classify and analyze failures [6]. We constructed prompts to extract the intent [6], nature [6], impacts [20], and lessons learnt [58] from the failures. The options for each dimension are illustrated in the descriptions in Table 3. The options are also illustrated in the prompts in §10. For each dimension, the LLMs were given the content of these articles and asked to classify each failure using the prompts in §10.

---

1We call these failures, rather than “compromises”, because some cases led to incidents and others were vulnerabilities that were apparently not exploited. See §2.3.

2The analysts were undergraduate and graduate students in computing, plus one faculty member.
Table 3: Dimensions used to analyze the capabilities of LLMs. The CNCF catalog includes "Type of compromise". Our research team labeled each catalog entry for the next three dimensions. The final dimension was assessed via a Likert scale.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of compromise</td>
<td>What kind of failure occurred [13] ? See Table 1 for types.</td>
</tr>
<tr>
<td>Intent</td>
<td>Was the &quot;software root cause&quot; of the failure, accidental or deliberate? [6]</td>
</tr>
<tr>
<td>Nature</td>
<td>Was the failure a vulnerability or an exploit? For exploits, was the actor an insider or outsider? [6]</td>
</tr>
<tr>
<td>Impacts</td>
<td>What kind(s) of impact resulted? The options are taken from [20]: (1) Data or financial theft, (2) Disabling networks or systems (3) Monitoring organizations or individuals, (4) Causing physical harm or death (5) All of the above are possible (6) Unknown or unclear.</td>
</tr>
<tr>
<td>Solutions/learnings</td>
<td>What was the quality of the solutions/learnings from the failure, that the LLM provided [58]?</td>
</tr>
</tbody>
</table>

4.3.2 For RQ2. For RQ2, we opted not to build a controlled taxonomy of "lessons learned" due to the open-ended nature of the prompt. Instead, we had human raters evaluate the recommendations using a 5-point Likert scale, ranging from "Strongly disagree" to "Strongly agree". The humans rated the LLM’s response in relation to the quality of the LLM’s response and whether it would mitigate a future attack. We had the same 3 pairs of 2 analysis evaluate the LLM’s output and the average score of each pair of taken as the final rating for the LLM’s response for each article.

4.4 Automated approach: LLMs

4.4.1 LLM selection. We used two popular, state-of-the-art LLMs that are publicly available at time of writing (June 2023): OpenAI’s ChatGPT model [76] and Google’s Bard model [71]. Their properties are summarized in Table 5. Other large language models are available, e.g., Claude [5] and Cohere [15], but GPT and Bard are the most widely used due to their user-friendly interfaces.

ChatGPT-3.5-turbo, OpenAI’s LLM. GPT-3.5-turbo is a large language model created by OpenAI. It uses a deep learning architecture known as a transformer [86]. It is currently one of the most popular and accurate LLMs [91]. GPT-3.5 uses 175 billion parameters and is trained on the same datasets used by GPT-3 but with a fine-tuning process called Reinforcement Learning with Human Feedback (RLHF) [56].

Bard, Google’s LLM. Bard is another popular and accurate LLM created by Google. Bard also uses transformers. It uses an optimized version of Language Models for Dialogue Applications (LaMDA) and was pre-trained on a variety publicly available data [57] including dialogue [37].

4.4.2 Prompt engineering. A prompt is the specific query (instructions or questions) given to an LLM. The behavior of an LLM varies widely as a result of seemingly minor tweaks to its prompt [55]. Prompt engineering is the process of crafting a prompt for an LLM to increase the quality of its response [89].

We used prompt engineering to iteratively develop prompts. We referred to various studies on prompt engineering [64, 89, 90]. For each dimension, we refined the prompt by issuing a basic query, then applying each prompt engineering technique in a cumulative sequence until the performance peaked, preserving any changes that improved from the best observed performance. Table 6 describes our approach using the first dimension, "Type of Compromise", as an example. This prompt engineering phase was conducted...
4.5 Experimental Setup

4.5.1 Order of prompts. We prompted LLMs in the order of Table 3.

4.5.2 Parameterization of LLMs. We focused on the two primary adjustable parameters of GPT-3.5, namely temperature and top_p, as outlined in Table 5. According to the OpenAI documentation, when one of the parameters is tuned, the other should be maintained at its default setting [65]. Our preliminary tests, as shown in Table 6, were conducted with a temperature of 0 and a default top_p value of 1.

After finalizing the prompt, we examined the effect of the parameters on accuracy for the "Type of compromise". For this article, accuracy decreased as the temperature increased. The accuracy was 78% at a temperature of 0, which declined to 64% at a temperature of 0.5, and further reduced to 50% at a temperature of 1. A similar trend was noted for the top_p parameter.

The optimal performance, with an accuracy of 78%, was achieved with a temperature of 0 and the top_p parameter at its default value of 1. We retained these parameter settings for the remainder of our analysis. This decision aligns with the guidelines provided in OpenAI’s documentation [65], which suggests that a lower temperature results in more focused and deterministic responses, a characteristic that is beneficial for article analysis.

4.5.3 Number of trials. We noted that the responses of GPT-3.5, configured with Temperature=0, exhibited consistent behavior. Consequently, a single trial was conducted to evaluate GPT’s accuracy across the dataset. Bard’s responses were less consistent, but the rate limit was low so we could only conduct one trial.

4.6 Data Analysis

We compared the results of the manual analysis against the automated analysis by the LLMs.

Table 5: Specifications of the LLMs used in the evaluation: GPT-3.5 and Bard. GPT’s tuning knobs use a 0-1 scale.

<table>
<thead>
<tr>
<th>Model</th>
<th>Cost-to-access</th>
<th>Rate limit</th>
<th>Parameters</th>
<th>Tuning knobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT-3.5-turbo-16k [66]</td>
<td>Input-$0.003/1K tokens, Output-$0.004/1K tokens</td>
<td>16K tokens per prompt</td>
<td>175 billion</td>
<td>Temperature: Higher values mean greater randomness of the new predicted word. Default: Unclear. top_p: Nucleus sampling. Model considers the results of tokens with top_p probability mass. top_p = 0.2 means when predicting the next work consider only tokens in the top 20% probability mass. Default: 1.</td>
</tr>
<tr>
<td>Bard [71]</td>
<td>Free</td>
<td>Unknown (estimate: 2K tokens per prompt and 50-100 prompts per 9 hours) [35]</td>
<td>137 billion</td>
<td>None available to users</td>
</tr>
</tbody>
</table>

on a subset of 20% of the dataset; we used the most recently published articles from the catalog as of June 2023. The final version of each prompt is available at §10.

For RQ1, we treated each LLM as another analyst and found how accurate it is at classifying various dimensions. We quantitatively report the LLM’s accuracy to measure its correctness for each dimension of analysis. In cases where the LLM’s analysis disagreed with the manual analysis, we examined its justifications. We qualitatively report some of our observations.

For RQ2, many distinct “lessons learned” are possible. We had analysts review each article and then the recommendations by GPT. The analysts rated the recommendations on whether the recommendations were appropriate, to the article on a 5-point Likert scale: "Strongly disagree", "Disagree", "Neither disagree nor agree", "Agree", and "Strongly agree". We did not experiment with Bard for this research question due to its rate limits.

5 RESULTS AND ANALYSIS

5.1 RQ1: How effective are LLMs replicating analysis of SW supply chain failures?

Table 7 summarizes the accuracy of GPT and Bard for the type of compromise, intent, nature, and impacts. GPT consistently outperformed Bard. We therefore focus our detailed analysis on GPT.

For most articles, GPT performed well on most dimensions. As depicted in Figure 4, GPT demonstrates an accuracy exceeding 75% (indicating correct responses in three out of four dimensions) in the majority of instances (62%).

When the manual raters had higher agreement, GPT tended to agree with them. GPT had high accuracy in the “Intent” and “Nature” dimensions, with accuracies of 88% and 74%, respectively. These dimensions exhibit Cohen’s $\kappa$ values of 0.87 and 0.58, respectively (Table 4), demonstrating substantial agreement between the analysts. In the “Impacts” dimension, the LLM produced an accuracy of 52%, as indicated in Table 7. The Cohen’s $\kappa$ was also low, at 0.34, as shown in Table 8. We conjecture that GPT agrees with analysts when there is a consensus amongst analysts regarding the labeling.

GPT had trouble when offered multi-answer as an option. For example, for the “Impacts” dimension it could choose from 4 specific impacts, or “All of the above/Multiple”, or “Unknown/Unclear”. In 87% of the cases, raters chose one of the multi-answer options, while GPT chose one of the specific options. GPT only selected “All of the above” three times and “Unknown/Unclear” once. We conjecture that when GPT was uncertain about the impacts, it opted for the most probable outcome of software supply chain failures in
Table 6: Techniques used to improve the prompts, illustrated for the prompt associated with the dimension of type of compromise. 

<table>
<thead>
<tr>
<th>ID</th>
<th>Technique</th>
<th>Prompt</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial prompt without any techniques</td>
<td>&quot;Classify the attack from the following choices Choice 1: Dev Tooling Choice 2: Negligence Choice 3: Publishing Infrastructure Choice 4: Source Code Choice 5: Trust and Signing Choice 6: Malicious Maintainer Choice 7: Attack Chaining Based on the information provided in the Articles. Article: {article} &quot;</td>
<td>33</td>
</tr>
<tr>
<td>1</td>
<td>Providing context/definitions- adding definitions of the options (improving upon ID: 0)</td>
<td>&quot;Classify the attack from the following choices Choice 1: Dev Tooling- Occurs when the development machine, SDK, tool chains, or build kit have been exploited. These exploits often result in the introduction of a backdoor by an attacker to own the development environment. Choice 2: Negligence- Occurs due to a lack of adherence to best practices. TypoSquatting attacks are a common type of attack associated with negligence, such as when a developer fails to verify the requested dependency name was correct (spelling, name components, glyphs in use, etc). Based on the information provided in the Article, Article: value&quot;</td>
<td>+36 (69)</td>
</tr>
<tr>
<td>2</td>
<td>Reflection Pattern- asking the LLM to explain its answer (improving upon ID: 1)</td>
<td>Adding the sentence &quot;Explain your answer using the given definitions and return the option.&quot; Before passing the article.</td>
<td>+2 (71)</td>
</tr>
<tr>
<td>3</td>
<td>Template technique (JSON format) and adding delimiters (improving upon ID: 2)</td>
<td>&quot;Use JSON format with the keys: 'explanation', 'choice'. Based on the information provided in the Article delimited by triple backticks. Article: <code>\{article\}</code> in the end.</td>
<td>+7 (78)</td>
</tr>
<tr>
<td>4</td>
<td>Placement of article- placing the article on top (improving upon ID: 3)</td>
<td>Based on the information provided in the Article delimited by triple backticks. Article: <code>\{article\}</code> &quot;Classify the attack from the following choices&quot;</td>
<td>-14 (64)</td>
</tr>
</tbody>
</table>
| 5  | The Cognitive Verifier Pattern- asking the LLM to generate addition questions to help it find the correct answer (improving upon ID: 3) | "... Choice 7: Attack Chaining- Sometimes a breach may be attributed to multiple lapses, with several compromises chained together to enable the attack. The attack chain may include types of supply chain attacks as defined here. However, catalogued attack chains often include other types of compromise, such as social engineering or a lack of adherence to best practices for securing publicly accessible infrastructure components. Generate two additional questions that would help you give a more accurate answer. Combine them to produce the final classification. Do not return these questions. Explain your answer using the given definitions and return the option. Only return JSON format with the keys: 'explanation', 'option' Based on the information provided in the Article delimited by triple backticks. Article: ```\{article\}

```

" | -14 (64) |
| 6  | Adopt a persona- asking the LLM to look at the article from an expert perspective (improving upon ID: 3) | Act as an software analyst and classify the attack from the following choices ... | -21 (57) |
| 7  | Citing evidence- asking for evidence from the text (improving upon ID: 3) | Explain your answer using the given definitions and return the option. Give evidence from the article to back up your answer. Use JSON format with the keys: 'explanation', 'option' | -14 (64) |

Table 7: Total accuracy over all the articles for each LLM.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>GPT</th>
<th>BARD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of compromise</td>
<td>59%</td>
<td>28%</td>
</tr>
<tr>
<td>Intent</td>
<td>88%</td>
<td>88%</td>
</tr>
<tr>
<td>Nature</td>
<td>74%</td>
<td>69%</td>
</tr>
<tr>
<td>Impacts</td>
<td>52%</td>
<td>45%</td>
</tr>
</tbody>
</table>

We observe that for the articles where the “Type of compromise” (ground truth provided by CNCF), we sometimes agreed with GPT over the CNCF. Figure 5 represents the distribution of GPT’s choice and when they were incorrect according to the CNFC ground truth. We examined the 14 articles where both the type of compromise and impacts were incorrectly identified. For these instances, two raters with an inter-rater agreement, \(\kappa\) of 0.82 found that most of the time, if they disagreed with CNCF, they concurred with GPT and vice versa. In the 8 instances where raters disagreed with CNCF, they agreed with GPT 6 times; the same ratio was observed when these articles (which focus on IT software). That option is data and financial theft, which it chose 49 times out of 65.
they disagreed with GPT and agreed with CNCF. For 2/14 articles they disagreed with both GPT and CNCF.

5.2 RQ2: Do LLMs suggest viable mitigation strategies for preventing future failures?

To address our second research question, we asked raters to evaluate GPT’s proposed solutions/learnings using a 5-point Likert scale. The average ratings are depicted in Table 8. The mean score across all three questions is 3.83. The raters generally held a positive or neutral view of GPT’s “Lessons learned”; 42% of the ratings were above 4 (agree), and only 5% of the ratings fell below 2 (disagree).

For further analysis, we randomly selected two articles where the average score of both the raters > 4, and two where < 2. See §10 for the full “Lessons Learned” for these cases.

**Factors for strong ratings (average score ≥ 4).** We believe the LLM demonstrated good performance in these cases due to the depth of the articles. Article 7 [10] describes the PHP Supply Chain Attack on Pear, and includes technical details of the failures, the exploitation method, and the patch. GPT utilizes the information provided in the blog, combined with its own knowledge, to suggest suitable solutions, e.g., “encouraging the use of Intrinsic or similar Node.js packages to whitelist and control access to sensitive resources and APIs”.

**Factors for weak ratings (average score ≤ 2).** We believe the LLM demonstrated poor performance in these cases because the articles had few details. Articles 65 [17] and 67 [74] are brief and lack substantial technical details of the failures. Article 67 discusses remote exploitation of a Gentoo server and mentions ongoing forensics. It primarily serves as a notice to users. Article 65 discusses the backdooring of WordPress but provides little information that could inform solutions/learnings. The advice given by GPT is hence generic, such as “investigate the incident and address the vulnerability” and “conduct code audits”.

<table>
<thead>
<tr>
<th>Question</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is the advice helpful in general for software supply chain failures?</td>
<td>3.72</td>
</tr>
<tr>
<td>Is the advice related to the specific failure mentioned in the article?</td>
<td>4.15</td>
</tr>
<tr>
<td>Can the advice be used to solve/mitigate the failure mentioned in the article?</td>
<td>3.62</td>
</tr>
</tbody>
</table>

6 DISCUSSION

**Is using LLMs worth it in this context?** We found that both LLMs in our experiment were capable of simpler forms of analysis, such as distinguishing whether a vulnerability was actually exploited. However, for more complex questions that require some amount of context or judgment, neither LLM achieved a high level of agreement with the CNCF analysts or our manual raters. We believe the current generation of off-the-shelf LLMs does not offer a high enough level of agreement with expert judgment to make it a useful assistant in this context.

One potential path to improving performance is fine-tuning the LLM using baseline knowledge such as this catalog, and then applying it on future issues [22]. An alternative is to integrate a domain-specific NLP model fine-tuned on cybersecurity data (e.g., CyBERT [73]), which might help on this specialized task. We emphasize that CyBERT is not an LLM with general Question-Answer capability, so integrating it would require non-trivial design and implementation work.

**Will LLMs be a viable alternative to manual analysis in the future?** In the past few years, OpenAI’s GPT models have advanced from simple tasks (GPT-1, GPT-2) to the performance reported here (GPT-3.5). The recent GPT-4 model is more impressive still [25]. We expect the next generation of LLMs will be suitable aids or replacements for this class of manual analysis.

**Future Work.** The scope of this analysis could be broadened to encompass additional LLMs, such as Claude [5] and Cohere [15], as well as to incorporate cybersecurity-specific NLP tools such as CyBERT [73]. Additional prompt engineering, and tailoring the prompts per LLM, might improve the accuracy of the results. Lastly,
We evaluate the ability of Large Language Models (LLMs) at characterizing software supply chain failures. Our study revealed that LLMs are particularly effective when manual analysts are able to reach a consensus on the characteristics of the failure. In contrast, their performance tends to deteriorate when the agreement among raters is low. The quality of the LLMs’ outputs also depends on the level of detail provided in the source articles, with more comprehensive articles leading to higher-quality responses. We conjecture that while LLMs offer a valuable tool for rapidly analyzing large volumes of text, they have not yet reached a stage where they can replace human analysts or manual classification. Rather than viewing LLMs as a replacement for human input, they should be considered as a supplementary tool that can assist human analysts. As the depth of detail in postmortems and articles increases, and as LLMs continue to improve, they may evolve into viable analytical resources.

9 ACKNOWLEDGMENTS

This work was supported by Cisco, Google, and NSF award #2229703.

Per the Association for Computing Machinery’s policies on authorship: OpenAI’s ChatGPT model (v4) was used during manuscript preparation. Prompt: Can you make the following clearer? “TEXT SNIPPET”. We reviewed answers to ensure it did not change the ideas.

10 DATA AVAILABILITY

Our data, analysis, and code is available at https://doi.org/10.5281/zenodo.8365116.

REFERENCES


