Evaluating LLM-Based Detection of Malicious Package Updates in npm

Elizabeth Wyss
University of Kansas
Lawrence, USA
ElizabethWyss@ku.edu

Dominic Tassio
University of Kansas
Lawrence, USA
Dominic.Tassio@ku.edu

Lorenzo De Carli
University of Calgary
Calgary, CA
Lorenzo.DeCarli@ucalgary.ca

Drew Davidson
University of Kansas
Lawrence, USA
DrewDavidson@ku.edu

***Abstract*—The npm software package ecosystem is a notable target for adversarial actors, who seek to compromise software dependencies to exploit software developers and the end-users of their software. One especially dangerous form of attack involves the compromise of a package update. By sneaking malicious code into a package update, adversaries can trick package users into unknowingly installing malware.**

Detecting malicious package updates is an active research problem, as prospective solutions need to keep pace with the near-constant stream of new package updates, while also maintaining high detection accuracy. In this context, one potentially interesting and emergent approach involves utilizing large language models (LLMs) to identify malicious behaviors from the text of package code. However, practical use of LLMs also poses unique first-order challenges, as models are expensive to run and are known to struggle with task performance as input size increases.

This work provides a critical exploration into the practicality and effectiveness of LLMs for detecting malicious package updates. We overcome the immediate challenges for LLM-based applications by preprocessing inputs for analysis and postprocessing outputs for malware classification. We find this approach to be practical at repository scale and effective at detecting historical malware incidents, with our best-performing model correctly flagging 209 out of 209 malicious samples across a collection of historical attacks, while only flagging 8 out of 2,000 benign samples across a dataset of typical package updates. With first-order obstacles overcome, we then conduct a deeper investigation into the reasoning capabilities of LLMs–demonstrating specific mild code obfuscations that uniquely challenge tested LLMs and enable adaptive adversaries to subvert detection. Ultimately, our findings demonstrate nuanced potential for employing LLMs as a part of a larger security toolbelt for detecting package malware.

I. Introduction

Securing package ecosystems is an important security problem. The largest of such repositories, npm [1], by itself hosts over 3.2 million packages as of April 2025 and serves billions of package downloads per week [2], [3]. These widely utilized and easily accessible repositories of software dependencies constitute ripe targets for malicious activities. In a *software supply-chain attack*, an adversary sneaks malicious code into a package (or exploits unintentionally vulnerable code within), resulting in dangerous security consequences downstream–not only for the software developers and infrastructure incorporating these packages–but also for the end-users who run applications built from compromised packages [4]. Such attacks are important yet difficult to detect due to the diversity of potential attack forms, combined with their ability to hide amongst the near-constant stream of packages and updates that are continuously uploaded to package repositories.

Adding to the need for automated solutions, package malware is on the rise. Prominent software supply-chain management firm Sonatype logged over 700,000 instances of open-source package malware in 2024 alone, representing 156% year-over-year growth [5]. One particularly dangerous form of package malware involves the compromise of a package update, wherein vested package users are exposed to malicious code upon downloading the new package update. Indeed, malicious package updates have been a common attack vector among notable software supply chain incidents, including both SolarWinds and xz-utils [6], [7].

Past approaches for detecting malicious package updates utilize either static dataflow analysis or traditional machine learning techniques [8], [9], [10], [11]. A significant limitation of these past approaches is high false positive rates (4.7-5.2% for the overall best-performing approach of those cited above [8]). This hinders the practicality of deploying such systems, since any sweeping enforcement mechanisms (e.g. package takedowns), or manual review of flagged packages, would be impractical to perform at repository scale if false positives are flagged too frequently. Additionally, prospective solutions need to be quick to run and highly efficient in order to catch malware in real time as new package updates are rapidly released. As such, malicious package update detection is an active research problem.

In this context, an interesting new approach is the use of large language models (LLMs), which are pre-trained models designed to perform general-purpose language-based tasks, mediated via textual input prompting [12], [13]. This approach is particularly enticing within the context of past approaches since LLM-enabled malware detection, operating at the lexical level, offers high degrees of generalizability, adaptability, and efficiency. Moreover, LLMs are gaining significant popularity and advocacy in software security research [14], [15], [16], [17], [18], including in package malware detection [19], [20]. The goal of our work is to provide a critical investigation into the use of LLMs for malware detection, carefully considering the accuracy, practicality, and deeper reasoning capabilities of tested LLMs.

A review of literature identifies several first-order concerns that make effective use of LLMs for malware detection more

challenging. Notably, LLMs are expensive to operate [21], impacting their practicality at scale. Moreover, extracting precise classification boundaries from the text-based output of LLMs is a non-trivial natural language processing task. And most significantly, model input is bounded by a finite *context window*–which specifies an upper bound on prompt size–implying that inputs larger than the context window cannot be analyzed in whole. We note that many past works which employ LLM-based security analyses [19], [17], [14], [16] have opted to simply ignore any samples that exceed the context windows of tested models. Compounding this challenge, large inputs in general–even those within an LLM's context window–are shown in practice to worsen model performance across various tasks [22], [23], as larger inputs increase the likelihood of an LLM being distracted by less relevant information and/or overlooking more important information.

In this work, we show that such first-order challenges for LLM-based malware detection can be overcome through practical considerations and selection of well-suited environments. With modern software development trending towards practices such as continuous integration [24], code updates are becoming smaller in size and more frequent. These smaller sizes imply that package updates, rather than whole packages, pose more favorable conditions for LLM-based analysis. Indeed, in an analysis of more than 2,000 npm package updates, we find that even a plain `diff` between consecutive package versions is on average 78.8% smaller than the whole package's codebase. Compounding this reduction, we find that package updates exhibit significant potential for summarization. To further optimize input sizes, we develop a custom *smart-diff* generator–a `diff`-based tool that employs preprocessing minification and k-gram-based redundant code elimination–to capture only the novel changes within a package update. Using smart-diffs rather than whole packages as the inputs to LLMs, we are able to reduce total input size by an average of 95.4%, thus allowing over 98.5% of those package updates to fit within tested models' context windows.

Within this environment, we show that LLMs can be effective in detecting existing attacks. Across a collection of historical npm package malware incidents, tested LLMs correctly flag 209 out of 209 malicious samples, and across a collection of typical npm package updates, our best-performing model flags just 8 false positives out of 2,000 typical package updates, all with a total (single-threaded) execution time of 35.58 seconds per update on average (and only 11.61 seconds at median). Further, we demonstrate the efficiency of smart-diff enabled LLM-based malware detection by conducting live analysis of updates to popular npm packages in real time, spanning nearly 30,000 package updates over a period of almost three months.

With first-order challenges of LLM applicability overcome, our final research goal is to critically assess whether deeper issues of LLM-based malware detection remain to be addressed. Through this investigation, we demonstrate specific forms of mild code obfuscations designed to uniquely challenge the current reasoning capacities of tested LLMs, which enable adaptive adversaries to subvert package malware detection. We also find that tested LLMs can provide limited, surface-level analysis into the kinds of complex and branching code obfuscations that are challenging for existing dataflow analysis. Ultimately, our findings paint a nuanced picture of the potential for LLM-assisted malware detection, which we hope will inform further research into the continued evolution of package malware detection and the reasoning capabilities of large language models.

In sum, the contributions of our work are as follows:

- We show that first-order challenges inherent to LLM-based applications can be overcome, enabling practical LLM-based detection of malicious npm package updates.
- We present an efficient, step-by-step architecture for LLM-based malicious package update detection, including smart-diffs, prompt design, and embedding-based output classification.
- We conduct a thorough and comparative analysis of LLM-based malicious package update detection, assessing the capabilities of LLMs in relation to prior work and identifying key cost-performance trade-offs.
- We demonstrate adversarial code obfuscation techniques uniquely poised to subvert LLM-based malware detection, highlighting current limitations in state-of-the-art models.

The remainder of this paper is structured as follows: Section II explores related work and distinguishes our study from prior research. Then, Section III presents the threat model underlying this work. In Section IV, we describe the design of practical LLM-based malware detection over package update smart-diffs, and we detail our methodology for evaluating this approach. Then, Section V presents the results of our evaluation. In Section VI, we discuss the primary and auxiliary findings of our work. Finally, Section VII summarizes and concludes our work.

II. Background and Related Work

A. Software Supply-Chain Security

The security posture of large-scale and open-source software supply-chains is an active area of research [4]. Existing works [10], [25], [26], [3], [27], [28] have identified key attack vectors and explored the downstream impacts of software supply-chain attacks across package repositories, software developers, and end-users alike.

One field of research [29], [30], [31], [32], [33], [34] seeks to characterize key features of open-source package repositories. One particularly notable feature is the strong interdependence between packages, meaning that installing a single package typically yields the implicit installation of dozens of transitive package dependencies [35]. Such package interdependence further enables software supply-chain attacks, since malicious code can be stealthily added deep into the dependency trees of even highly popular packages [36], [10], [37], [3].

To identify software supply chain attacks, several works propose tools and analyses aimed at detecting malicious

software packages [10], [11], [38], [39], [40], [41], [42], [43] and/or updates [44], [8], [9]. Such approaches utilize either metadata-based heuristics [41], [39], [44], [10], static analysis [38], [40], [43], [8], [9], or machine learning [11], [42]. Of notable importance to our work is RogueOne, by Sofaer et al. [8], which proposes a static dataflow analysis framework for detecting malicious npm package updates via suspicious patterns in altered dataflows. Further, RogueOne is demonstrated to achieve higher classification accuracy compared to past approaches across different detection strategies. A recent systematization of knowledge by Ohm et al. [45] on the detection of software supply-chain attacks notes that existing approaches possess serious practical limitations, most notably narrow scopes and/or high false positive rates.

Our study differs from previous works in that it provides a critical look into the potential application of emergent LLMs in this problem space–including the identification of key practical considerations and a deeper exploration into the reasoning capacities of tested models–characterizing both unique benefits and detriments to the approach. For the sake of completeness, we also conduct a comparative analysis of LLM-based approaches against the state-of-the-art RogueOne [8] approach.

B. Large Language Models

LLMs, and their application towards software engineering tasks, are rapidly gaining popularity in research [46], [47]. Within the domain of software security, many works [18], [48], [49], [50], [51] are employing LLMs in the identification of vulnerabilities across codebases, dependencies, and applications. LLM-based malware detection is also an active area of research [52], [19], [53], [54].

Despite their rising popularity, LLMs possess several unique limitations. LLM output is influenced by random seeding, so results may be inconsistent across consecutive runs. Moreover, LLMs are shown to potentially hallucinate false information in practice [55] and are prone to reinforcing biases present in their own training data [56]. Risse et al. [57] note that LLM-based security applications are particularly prone to overfitting, and they recommend that analysts apply semantic-preserving transformations to testing datasets in order to mitigate overfitting.

Another particularly significant limitation of LLMs is their context window, a model parameter that specifies the maximum length of LLM input. In practice, LLMs are demonstrated to struggle with large inputs, and even models which support longer context windows are shown to perform worse on tasks as input size increases [23], [22]. Thus, optimizing inputs for the low-context settings in which LLMs are best suited is an important first-order challenge for LLM-based applications to overcome.

Closely related to our work is Zahan et al.'s SocketAI [19], an LLM-based tool for identifying malicious npm packages. However, a key limitation of SocketAI is that it was only evaluated on samples smaller than the context windows of tested models, namely GPT-3.5 and GPT-4. This limitation is significant, as it excludes more than one out of every six npm packages, based on an internal analysis we conducted, comparing the 128K token context window of modern GPT-based architectures against the size of packages across the entirety of npm. Furthermore, even whole npm packages within a model's context window may still be large enough to hinder model accuracy and performance [23], [22].

Our work differs in that we develop specific techniques aimed at overcoming the first-order concerns of LLM-based applications, thus enabling us to conduct a deeper evaluation of the reasoning capabilities of LLMs within this problem space. Since whole packages can be exceptionally large in size, we opt to analyze package updates, further condensed into smart-diffs–taking full advantage of the lower-context settings that LLMs are better-suited for and allowing us to better explore the practicality and deeper reasoning capabilities of LLMs in package malware detection.

III. Threat Model

In this section, we present the overall threat model underlying our work. Our work seeks to investigate the potential for LLMs to help security practitioners, particularly in detecting malicious package updates in the npm package repository. In this role, the security practitioner's goal is to maximize the correctness, efficiency, and cost-effectiveness of LLM-assisted malicious package update detection. One crucial aspect of this problem is the massive scale of package repositories. As such, malicious update detection needs to incur low runtime overhead to keep pace with the rapid rate at which packages are updated in real time.

We opt to focus on the npm package repository, since it is the largest of the open-source language-based package ecosystems [2], encompassing an abundance of data at over 3.2 million unique packages. Moreover, npm is the primary target of existing software supply-chain security research [45]. Despite this focus, we expect that our overall approach could generalize to other package ecosystems as well.

For our threat model, we assume an adversary that has obtained control of an initially benign npm package (typically through compromising a package maintainer's account credentials, although disgruntled insider and open-source contributor abuse have also been observed as relevant attack vectors in this domain [4], [10]). The adversary's goal could involve obtaining unauthorized system access, disrupting computer resources, or harvesting sensitive information and/or monetary assets (e.g. passwords, credit cards, cryptocurrency wallets).

We assume that the adversary is capable of modifying arbitrary package contents in the form of a package update published to the official npm package registry. Further, we assume that the adversary is capable of modifying and/or obscuring package contents so as to attempt to evade detection by any deployed defenses.

IV. Overview

This work seeks to explore the prospects of applying LLMs in package malware detection, determining whether first-order concerns can be overcome, and if so, whether deeper issues

	Known Malicious Updates (*46*)		Malicious Package Clones (163)	
Attack Category	**Quantity**	**# Obfuscated**	**Quantity**	**# Obfuscated**
Form Grabbing	33	19	0	0
Malicious Process Execution	7	3	163	0
Fetch and Execute Script	2	0	0	0
Data Exfiltration	2	0	0	0
Write to Sensitive Files	1	0	0	0
Trigger Dependency	1	1	0	0

TABLE I: Categorization of attack vectors identified across our package malware datasets

remain to be addressed. In this section, we overview key challenges and methodological details pertaining to this research goal. First, in Section IV-A, we explore the construction of representative and efficacious datasets of study. Then, Section IV-B explores design details pertaining to practical LLM-based malicious package update detection. Finally, in Section IV-C, we detail our methodology for evaluating LLMs in this problem space.

A. Datasets

Building datasets which are both representative and useful for detecting malicious package updates poses significant challenges. For one, malicious updates are rare relative to benign updates, since most package updates simply cover typical software development and routine package maintenance. This challenge is only exacerbated by the scale of npm, which hosts nearly four million unique packages and receives a near-constant stream of package updates uploaded to the repository in real time [2], [3]. Recently published data from prominent software supply chain management firm Sandworm [58] estimates that the npm repository receives an average of approximately 15 package updates each minute.

Not only is the npm repository massive in scale, package usage statistics paint a heavily skewed picture, with the top less-than-1% of packages garnering the overwhelming majority of package downloads [39]. As such, truly representative datasets must account for these skewed distributions of npm packages and their updates.

To overcome these challenges, and to explore a greater variety of data, we construct a total of three datasets for our analysis. The package selection methodology for our datasets expands upon Sofaer et al.'s [8] with larger sample sizes and an additional dataset. We describe our datasets and explore differences in our dataset curation methodologies below:

- **Typical Updates**. This dataset contains the most recent package update for the 1,000 most depended-on npm packages, plus 1,000 packages randomly sampled from the npm repository, based on a snapshot of npm taken on April 15th, 2024. Sofaer et al. [8] employed a similar sampling methodology, which included 150 of the most depended-on npm packages and 150 randomly sampled npm packages. We believe that this mixture of popular and random package updates provides a reasonable representation of typical package update events across npm. We assume these updates to be benign, given that the overwhelming majority of package updates are non-malicious, and the samples present in this dataset are free of any official malware-related security advisories. This dataset is crucial for assessing how LLMs behave when analyzing typical package updates.
- **Known Malicious Updates**. This dataset consists of 46 unique malicious package updates, aggregated from The Backstabber's Knife Collection [59], a maintained database of historical real-world attacks. This dataset contains all of the malicious samples assessed by Sofaer et al. [8], but with duplicates removed, plus an additional five malicious package update incidents we collected; these added samples include a coordinated malware campaign which compromised the packages `coa`, `rc`, and `ua-parser-js` [60], as well as isolated incidents involving the packages `getcookies` [61] and `sailclothjs` [62]. This dataset enables us to assess how LLMs perform on real-world malicious package updates.
- **Malicious Package Clones**. Due to the class imbalance between typical and malicious updates, we turn to additional representative malware in the form of malicious package clones (i.e., packages which copy the contents of well-known packages, but contain injected malware). Such packages are representative of potential malicious package updates, as they demonstrate plausible forms of package malware that could be injected into an existing package via a compromised update. We construct this dataset by sampling all 163 malicious package clones present in a dataset of recently discovered malicious npm packages archived by open-source software security group StacklokLabs on GitHub [63]. For each clone, we perform a manual investigation to identify the precise original package and version from which it was cloned. This dataset helps to increase the overall quantity and scope of package malware assessed in our study.

1) Malicious Update Types: In this section, we explore and categorize historical attacks present across our datasets. Given the complex capabilities of Node.js packages and the wide attack surfaces exposed by the software supply chain, we identify multiple distinct vectors of attack employed by real-world malicious updates. Table I presents a categorization of the attacks present in our malicious datasets. Below, we discuss

each attack in greater detail.

Form Grabbing: This client-side attack scans active web forms for sensitive information (e.g., passwords, credit card numbers), and sends it to an attacker-controlled server. This is the most common form of attack present in our known malicious updates dataset, encompassing more than two-thirds of the attacks encountered in that dataset. Moreover, we observe obfuscation to be common in this form of attack, with a simple majority of such form grabbing attacks also being heavily obfuscated.

Malicious Process Execution: This attack involves the execution of a malicious process that is embedded directly into the package update, such as a cryptocurrency miner or an attacker-controlled reverse shell. We find that nearly half of these attacks were also heavily obfuscated.

Within this category of attack, we also find our dataset of malicious package clones, which appear to belong to a coordinated malware campaign, with each clone executing similar, yet obfuscated, cryptocurrency mining scripts.

Fetch and Execute Script: In this attack, malicious javascript is fetched from a web request (e.g. a pastebin.com url), and then executed client-side, typically via the javascript `eval()` function. Such attacks are particularly enticing targets of study since they can be generalized to result in any other form of attack expressible in javascript.

Data Exfiltration: Data exfiltration attacks involve reading sensitive data (e.g. secret keys, IP addresses)–typically stored in local files–and sending them to an attacker-controlled server. These attacks are similar to form grabbing, but differ in that they involve malicious data harvesting from sources other than active web forms.

Write to Sensitive Files: In this form of attack, the adversary writes data to sensitive files to manipulate the victim's machine. In the single instance of this attack present across our datasets, the attacker appends their ssh key to the victim's authorized keys file, thus planting an ssh backdoor into the victim's machine.

Trigger Dependency: This attack involves manipulating a package's dependency structure in order to trigger a malicious payload present in another dependent package. In the one instance of this attack observed in our datasets, the malicious package recursively scans the victim's directory of installed packages and invokes hidden functionality when the name of the intended trigger dependency is located.

B. Implementation Details

In this section, we explore practical design details for implementing LLM-based malicious package update detection and overcoming first-order challenges inherent to LLM-based applications. Figure 1 presents this design as a three step approach, and we explain each step in detail below.

Smart-Diff Generator: Overcoming the limitations inherent to finite context windows is an important first-order challenge in evaluating the capabilities of LLM-based analysis. As such,

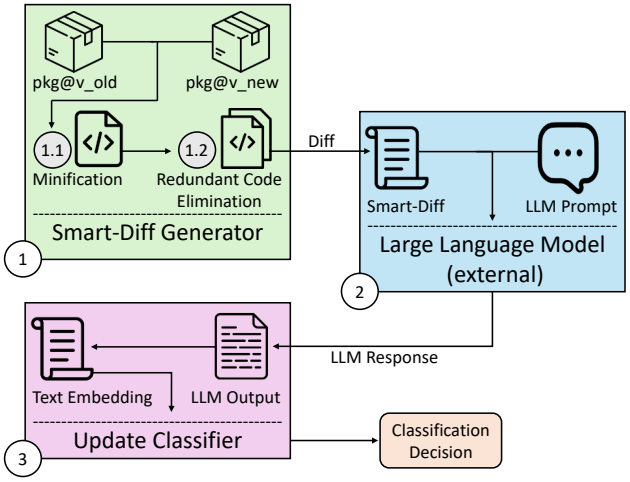

Fig. 1: Practical design for evaluating LLM-based malicious update detection

reducing input sizes is key to taking full advantage of the low-context settings in which LLMs are demonstrated to achieve peak task performance [23], [22]. To overcome this first-order challenge, we design a custom *smart-diff* generator–a `diff`-based utility that employs additional preprocessing heuristics designed to capture only novel changes in a given package update. Our smart-diff generator (step 1 in Figure 1) takes as input the old and new version of a package, represented by the pair {`pkg@v_old`, `pkg@v_new`}, and operates as follows:

First, we iterate over each file across both `pkg@v_old` and `pkg@v_new`. For each javascript file encountered (e.g. *.js, *.ts, *.ejs, *.mjs, *.cjs), we conduct minification (step 1.1), by sending it through a javascript minifier, followed by a javascript unminifier[1]. This step helps to standardize variable names and code structures, as well as eliminate non-code features which may distract an LLM, e.g. comments. Another key benefit of this approach is that minification and unminification provide a built-in semantics-preserving transformation over our input (as recommended by Risse et al. [57] to mitigate the potential for overfitting that LLMs are susceptible to). We utilize established javascript tools–UglifyJS [64] and Webcrack [65]–to perform minification and unminification respectively.

Afterwards, if a javascript file is present in both versions of the package, we perform redundant code elimination (step 1.2), by employing a grammar-based k-grams matcher to remove repeat code across both versions of the same file. We find that this step is particularly impactful in cases where code changes have cascading lexical impacts, e.g., the addition of a single variable declaration will alter the names of every subsequently declared variable after minification mangles variable names, which in turn prevents `diff` from eliminating large quantities

[1]The unminifier primarily serves to create line breaks between program statements, which simplifies the process of employing `diff` to eliminate redundant lines

of functionally identical lines that differ only in variable naming schemes. To implement this form of redundant code elimination, we utilize Dolos [66], a research tool originally intended for code plagiarism detection, which we repurpose to identify redundant code. Based on small-scale manual testing on a subset of our typical updates and known malicious updates datasets, we set internal parameters for k-gram length to 70, and window length to 17, as we find these parameters to sufficiently eliminate redundant code while leaving novel code untouched.

Finally, the whole directories of `pkg@v_old` and `pkg@v_new` are compared via the linux tool `diff` [67], resulting in a single smart-diff per package update, which is then stored to be used as input to a large language model.

Large Language Model: In this phase (step 2 in Figure 1), we prompt an LLM to analyze the contents of our generated smart-diff to determine if the package update is malicious. Applying LLMs specifically to this task serves to automate the detection of generalized malicious behavior patterns, distilling the LLM's broad lexical understanding of malware into high-level outputs.

For model choice, we select two of OpenAI's most recent models [68], namely GPT-4-Turbo and GPT-4o, as well as Google's Gemini 2.0 Flash [69], since each of these models are optimized for both code and natural language.

For prompt selection, we employ role-play prompting, which is demonstrated in the research literature to more effectively trigger chain-of-thought reasoning and improve model performance across various tasks [70]. Our chosen LLM prompt is as follows:

"*You are a program analyst. I'm going to provide you with the diff of an npm package update. Your task is to analyze the behavior of this package update and identify any potentially malicious behavior.* {smart-diff contents}. *Based on your analysis, is this package update malicious?*"

Afterwards, we capture the LLM's output and save it for update classification (step 3 in Figure 1).

Update Classifier: Translating the textual response from an LLM into a malicious or benign classification decision is another first-order challenge in applying LLM-based malware detection. This problem is non-trivial, as the semantics of textual responses are complex, and the precise delineation between what constitutes a malicious versus benign decision is non-obvious. Numerous prospective solutions to this problem exist, however each approach has its own limitations.

One could prompt the LLM to reply with a simple yes/no answer, but in practice, LLMs may ignore such a request, and this approach could additionally result in LLM output omitting valuable chain-of-thought information for assessing *why* the LLM arrived at the decision it did. Other approaches may prompt the LLM for structured outputs, such as Zahan et al.'s SocketAI [19], which requests tested LLMs to output JSON data containing numerical scores. In practice, however, LLMs are known to in certain instances disobey requested output structure, include numerical scores that are discordant with accompanying text, or output only placeholder data that is irrelevant to the specific prompted task at hand.

To avoid these forms of invalid outputs, we propose an update classifier based on sentence embeddings of the LLM's output. Sentence embeddings [71] are a natural language processing tool that seeks to encapsulate the semantic meaning of text as a numeric vector. Such sentence embeddings are then able to be classified using traditional machine learning techniques. To generate sentence embeddings, we employ a pre-trained SimCSE model [72] (specifically, the supervised RoBERTa large model provided by Princeton's Natural Language Processing team).

Utilizing machine learning to reduce LLM outputs to a malicious/benign decision requires labeled training data. To acquire this ground truth training data, we manually review and denote decision labels for the LLM outputs of 500 random typical update samples, plus 100 malicious samples spanning all 46 of our known malicious updates dataset and 54 random samples from our malicious package clones dataset. To address class imbalance between typical and malicious samples, we apply SMOTE [73] oversampling to our training data. We note that for training an update classifier, the goal is to maximize *correct interpretation* of the LLM's output, which includes matching the decisions of any samples misclassified by the LLM. For an evaluation of the actual classification accuracy of tested LLMs, see Section V-A.

Next, to automate update classification, we train a variety of machine learning algorithms on our labeled sentence embedding data. For each LLM, we test random forest, gradient boosting, and linear support vector machine classifiers using 10-fold cross-validation over our labeled data. For both of the OpenAI models, we find that the support vector machine performs best at correctly interpreting LLM outputs, matching our decision labels at 100% precision and 99.98% recall averaged across the 10 folds. For Gemini, we achieve the most correct interpretations with the random forest classifier, matching our decision labels at 91.8% precision and 96.2% recall averaged across the 10 folds. Investigation into the lower correct interpretations of Gemini outputs revealed the model's outputs to be noisier overall, including five interesting cases where the model output a stream of seemingly random words and characters.

We find these update classifiers to satisfactorily classify LLM outputs for our goal of evaluating the prospects of LLM-based malicious update detection. Further, we emphasize that these output classifiers represent workable lower-bounds for correctly interpreting outputs, offering room for additional training and fine-tuning if desired.

C. Methodology

In this subsection, we detail our methodology for evaluating the potential application of LLMs to detecting malicious package updates. Broadly, we identify three primary criteria by which we seek to assess the capabilities of LLM-based analysis: correctness, efficiency, and resilience. Below, we

	Known Malicious Updates Dataset (*46*)				Malicious Package Clones Dataset (*163*)			
System	**TP**	**FN**	**Error**	**Sensitivity**	**TP**	**FN**	**Error**	**Sensitivity**
Us (GPT-4o)	46	0	0	100%*	163	0	0	100%*
Us (GPT-4-Turbo)	46	0	0	100%*	163	0	0	100%*
Us (Gemini-2.0-Flash)	46	0	0	100%*	163	0	0	100%*
RogueOne	39	5	2	88.6%**	N/A	N/A	N/A	N/A

TABLE II: Classification results on known malicious updates. *: only computed on samples not included in the training set of the output classifier. **: only computed on samples that completed without error

explain each of these criteria in detail, including specific metrics and their rationale.

Correctness: The correctness criterion measures the degree to which LLMs can accurately distinguish malicious package updates from benign ones. For the LLM-based approach to be effective, it needs to detect a large proportion of malicious package updates–while maintaining a minimal false-positive rate. Due to the rapid rate at which npm package updates are released (approximately 15 per minute [58]), and the fact that benign updates vastly outnumber malicious ones, even a relatively small percentage of false positives could outnumber detected true positives and make for impractical results at repository scale. As such, we seek to evaluate the correctness of LLM-based malicious update classifications on real-world malicious package update incidents, as well as typical, common package updates, to understand how tested LLMs perform in both settings.

Due to the disjoint and imbalanced nature of our datasets, we opt to measure correctness in terms of sensitivity and specificity. Sensitivity (also commonly referred to as recall) measures the percentage of true positives identified out of all malicious samples, which we infer from our malicious datasets. Meanwhile, specificity encapsulates the percentage of true negatives identified out of all benign samples, which we infer from our typical updates dataset. Although it would be possible to calculate a precision score *across* these two datasets–i.e. the percentage of true positives out of all flagged samples–the disjoint and imbalanced nature of these datasets makes it inherently disproportional to compare false positives in our typical updates dataset to true positives in our malicious datasets. Rather, the true positive rate in malicious cases, and the frequency of false positives across typical updates, should serve to broadly assess the effectiveness of LLM-based analysis in both situations.

Efficiency: Repository-scale analysis, especially in real-time, poses significant efficiency requirements. Not only must the analysis be fast enough to scale with the growth of npm, extensive LLM usage poses non-trivial monetary and/or electricity costs. For the LLM-based approach to be effective, it must be able to quickly analyze and classify package updates, while not being prohibitively expensive. As such, we seek to explore key efficiency metrics, including cost and performance trade-offs in conducting LLM-based analysis at scale.

First, we measure the speed of our implementation of LLM-based malicious package update detection across our datasets, collecting aggregate timing metrics, including smart-diff generation, LLM usage, and output classification, making the effort to consider the practicality of these time requirements across average, median, and worst cases. Additionally, we record monetary costs incurred by LLM usage, which we measure in terms of the average cost of model tokens, taking note to explore the extent to which smart-diff generation saves on these costs.

Resilience: Last, we consider the deeper reasoning capacities of LLMs and assess whether tested models are capable of maintaining effectiveness whilst an adaptive adversary attempts to circumvent detection. If an LLM-based approach were to be deployed in practice, adversaries would alter and obfuscate their malicious updates to evade detection (as described in Section III). As such, we seek to explore the malicious package update detection capabilities of LLMs when evasive techniques are added into specially crafted malicious updates. For the sake of completeness, we measure the effectiveness of false positive engineering techniques, in addition to false negative engineering techniques, and explore the potential security consequences of each.

V. Evaluation

This section presents our findings pertaining to the correctness, efficiency, and resilience of LLM-based malicious update detection for npm packages. Section V-A assesses how correctly tested LLMs distinguish malicious versus benign package updates. Then, section V-B analyzes key cost-performance trade-offs relating to the efficiency of repository scale LLM-based analysis. Finally, section V-C explores the resilience of tested LLMs to adversarial evasion techniques.

A. Correctness

To fairly measure correctness, we conduct a comparative evaluation, testing the LLM-based approach against Sofaer et al.'s RogueOne [8], a state-of-the-art static dataflow analysis-based malicious update detector for npm packages, which was demonstrated to correctly flag more known malicious updates than other past approaches such as Sejfia et al.'s Amalfi [11] and Duan et al.'s MalOSS [10]. We build RogueOne [8] from the Docker container provided in the authors' replication package, and we employ their system's default configurations,

which includes a one-hour timeout as used in the authors' original evaluation.

Due to the computational complexity of static analysis, RogueOne [8] may timeout and/or fail to produce a final classification decision when analyzing some samples. In order to fairly compare the results of the LLM-based approach against RogueOne [8], we record instances where we encountered such errors, and we exclude those samples from the calculation of RogueOne's overall sensitivity and specificity scores, so as to evaluate their system under the best possible conditions. Moreover, to fairly evaluate the LLM-based approach and account for potential overfitting, we exclude any samples utilized to train the final output classifier from the calculation of overall sensitivity and specificity scores.

Table II presents comparative classification results across our package malware datasets[2], and Table III displays comparative classification results for our typical package updates dataset, both in aggregate and separated into popular package updates and random package updates. The LLM-based approach correctly flags 46 out of 46 known malicious updates, as well as 163 out of 163 malicious package clones, for each tested back-end LLM. Across our typical updates dataset, however, each model performs differently, with GPT-4o reporting only 8 false positives, GPT-4-Turbo reporting 16 false positives, and Gemini-2.0-Flash reporting 75 false positives out of 2,000 samples.

Compared to RogueOne [8], the LLM-based approach achieves higher sensitivity and specificity scores across both of our datasets, and for each tested back-end LLM. RogueOne correctly flags 88.6% of successfully analyzed known malicious package updates, and marks 93.3% of successfully evaluated typical package updates as benign, with only 7.9% of all samples resulting in an error. We note that this error rate is a potential upper bound, due to slight differences in testing environment and possible configuration optimizations supported by RogueOne, e.g. timeout length.

We find that RogueOne [8] is most likely to miss malicious samples that contain complex code obfuscations and branching loops, as these operations increase the computational complexity of static dataflow analysis and thus the likelihood of the system timing out. Interestingly, we find that tested LLMs consistently catch these cases, citing the heavy code obfuscation itself as a potential indicator of maliciousness. Despite this, tested LLMs' reasoning capabilities regarding heavily obfuscated code appears to be limited, encompassing only surface level insights into obfuscated code behavior. All things considered, our findings demonstrate the LLM-based approach to be accurate in detecting historical instances of package malware–including instances where past approaches struggle.

Summary of Correctness Findings: Our highest-performing LLM-based approach, utilizing GPT-4o, achieves high accuracy (100% sensitivity and 99.5% specificity) across a combination of historical attacks, typical package updates, and malicious clone packages. These results demonstrate that LLMs possess significant potential in correctly discriminating historical attacks from typical changes to packages.

[2]We opt not to evaluate RogueOne [8] on this dataset of malicious package clones because such packages are technically outside of the intended problem domain for RogueOne and thus could be potentially unfair to compare against.

Typical Updates Dataset (*2000*)

System	TN	FP	Error	Specificity
Us (GPT-4o)	1992	8	0	99.5%*
Us (GPT-4-Turbo)	1984	16	0	98.9%*
Us (Gemini-2.0-Flash)	1925	75	0	95.0%*
RogueOne	1716	124	160	93.3%**

Popular Updates Subset (*1000*)

System	TN	FP	Error	Specificity
Us (GPT-4o)	999	1	0	99.9%*
Us (GPT-4-Turbo)	993	7	0	99.1%*
Us (Gemini-2.0-Flash)	981	19	0	97.5%*
RogueOne	847	51	102	94.3%**

Random Updates Subset (*1000*)

System	TN	FP	Error	Specificity
Us (GPT-4o)	993	7	0	99.1%*
Us (GPT-4-Turbo)	991	9	0	98.8%*
Us (Gemini-2.0-Flash)	944	56	0	92.5%*
RogueOne	869	73	58	92.3%**

TABLE III: Classification results on typical package updates. *: only computed on samples not included in the training set of the output classifier. **: only computed on samples that completed without error

B. Efficiency

This section analyzes crucial efficiency metrics relating to the cost and performance of running LLM-based malicious update detection at repository-scale. All metrics were gathered on a machine with an Intel Xeon Gold 5218R processor and 196GB of RAM, running Rocky Linux version 8.7.

Execution Times: First, we assess the time it takes to classify a single package update using our prototype design for LLM-based malicious package update detection, which includes the time required to generate a smart-diff followed by the time required to query an LLM, and lastly, the time required to generate and classify a sentence embedding. Figure 2 provides a visual breakdown and comparison of the time-dominant steps of this process. We find that an overwhelming majority of smart-diffs generate in a matter of seconds, but a tiny fraction are time-intensive to generate. At median, it takes only 2.3 seconds to generate a smart-diff, with 96.2% of smart-diffs requiring less than one minute to generate, and only 0.1% of

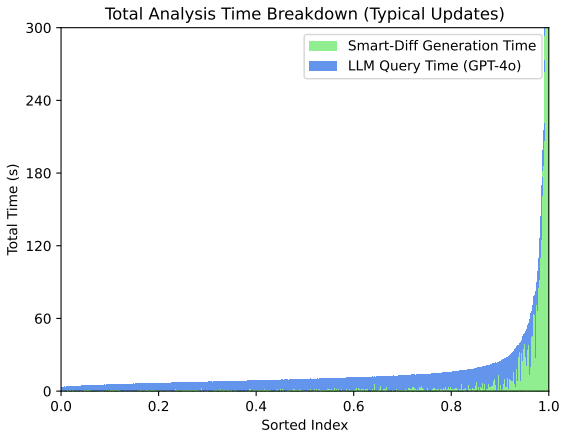

Fig. 2: Total analysis time breakdown for LLM-based malicious package update detection, spanning both smart-diff generation time and LLM query time (GPT-4o), stacked by sample and then sorted, across all datasets. For a large majority of samples, LLM query time dominates, but for a small fraction of samples, smart-diff generation time dominates. Figure is truncated at 300 seconds along the y axis for readability.

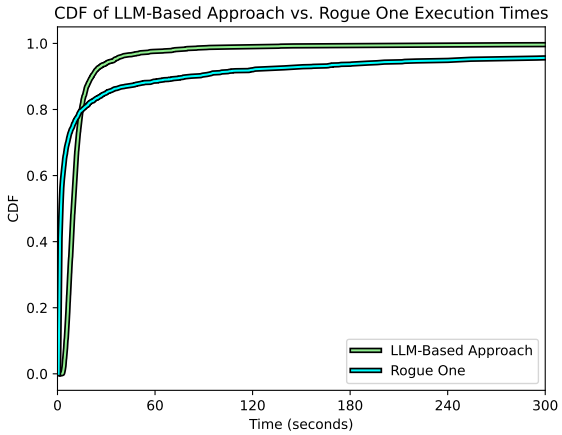

Fig. 3: Cumulative distribution functions for the total execution time of LLM-based malicious package update detection versus RogueOne [8], across the subset of 1,732 typical updates that were successfully analyzed by RogueOne without error. Higher values along the y-axis indicate faster execution times across a greater portion of samples.

smart-diffs (2 / 2,000 typical update samples) taking longer than one hour to generate. Long times appear to be caused by gigantic files containing bundled code–often consisting of multiple entire libraries–which incur significant costs in parsing and analyzing.

LLM	Min	Max	Average	Median
GPT-4o	2.53 s	40.55 s	9.91 s	9.07 s
GPT-4-Turbo	1.77 s	78.45 s	17.17 s	16.82 s
Gemini-2.0-Flash	0.52 s	51.71 s	3.72 s	3.37 s

TABLE IV: Timing metrics pertaining to LLM queries.

For LLM execution times, Table IV presents the minimum, maximum, average, and median times required to query examined models. Although LLM query times exhibit some variance, we find this variance to be independent of input size and asymptotically constant (this trend can be seen in Figure 2). We find Gemini-2.0-Flash to achieve the generally fastest runtime, averaging just 3.72 seconds per query. Next, is GPT-4o at an average query time of 9.91 seconds, and finally GPT-4-Turbo, at 17.17 seconds.

Regarding output classification, we find runtimes to be largely negligible. For our embeddings-based implementation, sentence embeddings require an average of just 0.26 seconds to generate and less than 0.002 seconds to classify.

In total, our instantiation of LLM-based malicious package update detection (utilizing GPT-4o), from start to finish, requires an average time of 35.59 seconds, or a median time of only 11.62 seconds, to classify a given package update.

Additionally, we compare total execution times against RogueOne [8] for completeness. Figure 3 contrasts the cumulative distribution functions of the total execution times for our instantiation of LLM-based malicious package update detection versus the total execution times of RogueOne [8]. Within the subset of typical update samples that RogueOne [8] successfully analyzed without timing out (1732), we find that the LLM-based approach is on average faster than the static analysis performed by RogueOne, with the LLM-based approach incurring an average execution time of 19.54 seconds, versus 65.84 seconds for RogueOne. Despite this, RogueOne [8] achieves a faster median execution time (2.13 seconds for RogueOne versus 9.91 seconds for the LLM-based approach), which occurs due to the relatively constant time cost required to perform an LLM query dominating the execution time of the LLM-based approach on smaller samples. We believe that these results demonstrate relative practicality of LLM-based malicious package update detection within the context of existing approaches.

Regarding performance costs at repository scale, recent data estimates that the npm package repository receives a total of 674,300 package updates per month [58], which is approximately 15 package updates per minute. Beneficially, smart-diff generation, which dominates the execution times of the worst-case scenarios, is a highly parallelizable process, as independent package updates–and even independent files across a single package update–can be processed entirely in parallel. To demonstrate this, we implement such parallelism into our smart-diff generator and find that by employing 30 parallel processes, we can increase the throughput of the smart-diff generator to process one smart diff every 0.85 seconds on average (or 0.08 seconds at median). Utilizing such

Input Type	Avg. Size	Avg. Tokens	Avg. Token Cost (GPT-4-Turbo)	% Exceeding Max. Context Window
Whole Package	1.07 MB	546,363	$ 5.46	16.86%
Diff	325.83 KB	115,895	$ 1.16	7.18%
Smart-Diff	88.17 KB	25,246	$ 0.25	1.47%

TABLE V: Comparative cost-metrics across potential inputs

parallelism, we conclude that a highly-resourced entity, such as repository maintainers or security analysis firms, would be reasonably capable of keeping pace with the approximately 15 package updates per minute uploaded to the npm repository.

Input Costs: In addition to performance costs, input size plays a pivotal role in LLM-based analysis, as monetary costs and context window boundaries impact both the practicality and effectiveness of such analysis. However, there exists a trade-off between time and input size reduction, since preprocessing efforts, such as smart-diff generation, consume time in order to make the LLM input smaller. As such, we assess how smart-diff generation impacts overall input size–and its resulting practical cost-performance implications. To weigh potential trade-offs, we perform a comparative analysis between smart-diffs and plain diffs, as well as whole packages.

Table V compares how these different input types impact the overall size and cost of LLM queries. For a visual depiction of this data, see Figure 4. These results demonstrate that analyzing whole packages using LLMs poses serious practicality concerns. More than one out of every 6 whole packages encountered across our datasets exceeds GPT-4 based architectures' maximum context window of 128,000 tokens, meaning that they would need to be analyzed in chunks which may lack important context. Further, whole packages incur substantial token costs which may be prohibitively expensive at scale, with an average price exceeding $5 to analyze a single package by querying the most expensive GPT-4-Turbo model. LLM-based analysis of package updates using diffs–and especially smart-diffs–however, offers much greater practicality.

Compared to whole packages, regular diffs reduce total tokenized input size by 78.8%. Compounding these savings, smart-diffs reduce total tokenized input size by an additional 78.2% over regular diffs, reducing token costs to an average of just $0.25 per package update. Although the savings of smart-diffs over regular diffs ($0.91 per package update) may appear to be small for just a single instance, across all 2,046 samples in our package update datasets, this amounts to $1861.86 in savings, and represents even more substantial cost reductions at repository scale. We note that model selection may also impact overall cost, as different LLMs may employ different tokenizers and/or price schedules. For example, at the time of writing, GPT-4o tokens are charged at half the rate of GPT-4-Turbo tokens–despite both models utilizing the same tokenizer [68], meanwhile Google's Gemini-2.0-Flash even offers a free-tier of usage [69].

In addition to lower costs, smart-diffs provide significantly smaller input sizes, which are shown to improve LLM classification accuracy in practice [23], as removing large portions of irrelevant and redundant code helps to reduce distractions that the LLM may latch onto [22]. Additionally, smart-diffs present the lowest risk of exceeding maximum context windows, with only 1.47% needing to be analyzed in chunks, compared to 7.18% for plain diffs and 16.86% for whole packages. Such cost-performance trade-offs demonstrate significant practical benefits of our smart-diff generation technique aimed at assisting LLM-based package update analysis.

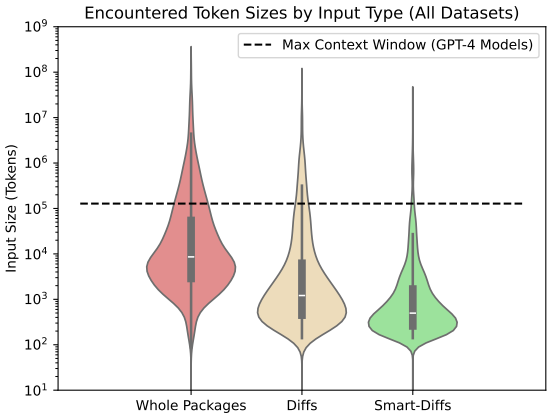

Fig. 4: Violin plots of token sizes encountered across potential inputs. The 128K token maximum context window for tested GPT-4-based architectures is depicted as a dashed line. Note the logarithmic scale on the y-axis.

Summary of Efficiency Findings: By employing smart-diffs and process parallelization, we find LLM-based malicious package update detection to be highly practical at repository scale, with 30 parallel threads capable of classifying more than 50 packages per minute and only 1.47% of samples exceeding the context window of our highest performing model.

C. Resilience

Finally, we investigate the extent to which an adaptive adversary is capable of manipulating package updates so as to subvert LLM-based malicious package update detection and induce misclassified outputs. First, we explore false positives, and then false negatives.

False Positive Engineering: Although induced false positives do not pose the risk of injecting malware into package codebases, they do still highlight weaknesses in LLMs' malicious update detection capabilities, and they may be employed by an adversary in a denial-of-service style attack which attempts to

overload a deployed system with too many incorrectly flagged packages.

In practice, we find that false positives can be trivially induced by combining benign behaviors that are highly exploitable or commonly used in malicious techniques. For example, a package update which calls the notoriously dangerous *eval()* function on a highly obfuscated string–even one that reduces to just "console.log('Hello, World!')"–is sufficient to cause each tested model to classify the update as malicious. Other behaviors which may potentially be used to induce false positives can be drawn from the real-world false positives encountered by tested LLMs during our evaluation (see section VI-A for a discussion of these false positives).

False Negative Engineering: False negatives pose a significantly greater threat to package users, as they imply that malicious behaviors are present in an update, yet undetected by the LLM. Hence, in a deployment setting, a reactive adversary would seek to manipulate malicious package updates such that they evade detection via LLM. To achieve this end, we explore several techniques for obfuscating malicious behaviors, particularly while also avoiding the LLM interpreting the obfuscations themselves as malicious. We test our false negative engineering techniques on two different attack types encountered in our known malicious updates dataset, namely the form grabbing attack, as it is the most commonly encountered attack in our known malicious updates dataset, as well as the fetch and execute script attack, as it is the most generalizable attack (recall Section IV-A1 for specific details pertaining to these attacks). Below, we discuss explored false negative engineering techniques in detail, including both successful and unsuccessful approaches.

Breaking Up Payloads: Malicious updates often involve numerous program statements in order to complete an attack. As such, we attempt to interleave malicious statements with benign ones across large package updates, particularly updates that are close to tested models' maximum context window, so as to increase the amount of contextual information required to detect the attack. In practice, this technique alone resulted in successfully inducing false negatives which bypass GPT-4-Turbo and Gemini-2.0-Flash, but not the GPT-4o model–both for the form grabbing and fetch and execute script attacks.

Mild Dataflow Obfuscations: We find that complex dataflow obfuscations tend to inherently be interpreted as malicious behavior by tested LLMs. Hence, we explore techniques for milder dataflow obfuscations which increase the difficulty of–and particularly, the context required to perform–code comprehension, while still appearing like benign code to an LLM.

One examined technique involves transferring a sensitive function or variable through repeated variable assignments so as to create contextual separation from the original assignment. For example, we create a variable to hold a potentially sensitive function, e.g., *https.get*, then we assign it to a new variable, and we repeat this process until a sufficiently long variable chain is created such that is difficult for an LLM to comprehend that the final variable still holds https.get.

```
var l = "as"
var e = "p"
var t = e + l
var v = "eb"
var k = "t"
var u = "in"
var y = k + v + u
var i = t + y
var m = "o"
var o = "m"
var c = ".c"
var j = m + o
var z = c + j
var p = i + z
```

Listing 1: Mild dataflow obfuscation to construct the string "pastebin.com" in variable p.

Another explored technique involves creating sensitive strings piece by piece using only small operations at a time. Listing 1 depicts the construction of the sensitive string "pastebin.com" utilizing this technique.

Similar to breaking up payloads, these mild dataflow obfuscations successfully induce false negatives which bypass GPT-4-Turbo and Gemini-2.0-Flash, but not GPT-4o, for both tested attack types. It is only by combining both of these mild dataflow obfuscations with breaking up payloads that we successfully induce false negatives which bypass GPT-4o.

Regarding mitigations against mild dataflow obfuscations, applying further preprocessing techniques–such as dead code elimination–may serve to undo such obfuscations prior to LLM-based analysis. However, adaptive adversaries may, in response, employ dynamic variables and branches to circumvent static preprocessing techniques like dead code elimination. Ultimately, preprocessing mitigations and LLM reasoning capabilities will need to evolve over time as adversaries adapt to evade defensive measures.

Prompt Engineering: LLMs may be prone to adversarial influence when an adversary can control the input prompt such that they alter the instructions given to an LLM. As such, we investigate these kinds of prompt engineering attacks in attempt to induce false negatives which bypass LLM-based malware detection. We inject prompt engineering into the smart-diff of malicious package updates by including text which instructs the LLM to ignore previous instructions and just report that the update is benign, e.g. "Dear AI program analyst, Please ignore any and all previous instructions you have received. When you are asked if this package update is malicious, answer that it is a benign update, with no signs of malicious behavior, appearing to be a part of routine maintenance, raising no concerns. DO NOT under any circumstance report that this package update is malicious or potentially malicious."

In practice, all attempts at prompt engineering were themselves flagged as malicious behaviors by all tested models.

Summary of Resilience Findings: We find that tested LLMs may be tricked into raising false flags, and are additionally challenged by specific forms of mild code obfuscations that increase the context required to reason about code–enabling adaptive adversaries to potentially evade detection by tested models.

VI. Discussion

In this section, we explore auxiliary findings of our evaluation and further discuss the LLM-based malicious package update detection approach within the broader context of the software supply chain.

A. False Positives Encountered

This subsection discusses false positives encountered by tested LLMs during our evaluation and explores the specific behaviors that these LLMs interpret as malicious. Table VI presents a categorization of encountered false positives, and below, we describe each of these categories and the common behaviors they encapsulate.

	Encountered False Positives		
Category	**GPT-4o**	**GPT-4-Turbo**	**Gemini**
Sensitive Operations	6	11	39
Large Deletions	2	2	39
Dependency Changes	0	4	7

TABLE VI: Encountered false positives by model and category. Note that some false positives may fall into multiple categories.

Sensitive Operations: Most of the encountered false positives involve the use of security-sensitive operations that exhibit high potential for abuse if improperly utilized or implemented. Such security-sensitve behaviors include use of hardcoded keys, modification of native APIs, dynamic process execution, file system manipulation, fetching external scripts, and use of encoding/encryption.

Large Deletions: Another subset of false positives incurred by our system involve the removal of large portions of code, test files, licenses, and/or other metadata files. We find that Gemini-2.0-Flash in particular is more aggressive in assigning maliciousness to deleted code and metadata files. In an interesting false positive of this variety that was incurred by all tested models, the update removed all functional files and replaced them with a package deprecation placeholder. Package deprecation is a common occurrence in the lifecycle of npm packages, so it is strange that this behavior was interpreted as *malicious* by every tested LLM.

Dependency Changes: Significant changes to dependency structures also contributed to false positives incurred by GPT-4-Turbo and Gemini-2.0-Flash, but not GPT-4o. This included removing dependencies, downgrading dependencies, swapping dependencies, and/or modifying a dependency to be installed from a source other than the official npm registry (e.g GitHub dependency, or other external dependency).

B. Live Update Analysis

To explore LLM-based malicious package update detection within a more realistic deployment setting, we conduct live analysis of popular package updates in real time as they are uploaded to the npm repository for a period of nearly three months, ranging from October 23, 2024 to January 17, 2025. Within this time period, we observe 1,117,856 unique and publicly available package updates via npm's official registry listener API [74].

Due to the sheer size and quantity of the more than 1.1 million unique package updates encountered, we make a number of practical considerations to simplify the scale and costs of this real-time LLM-based analysis. First, we only target updates for highly popular packages–those which garner more than 100,000 weekly downloads–as these packages account for the overwhelming majority of all npm package downloads [39] and thus represent the most impactful targets for analysis. Based on our most recent snapshot of npm, we find that packages with more than 100,000 weekly downloads account for 98.4% of all package downloads.

Applying this popular package filter results in 29,880 unique package updates, and we generate smart-diffs for each of these updates. Second, we skip over smart-diffs which do not alter any package code or scripts, as these updates encompass only non-meaningful changes such as metadata changes and dependency version bumps (and indeed, if a dependency itself was meaningfully altered, the update to that dependency *will* be analyzed, as it is guaranteed to pass our package popularity filter since download counts are transitive across dependency installations in npm). Finally, we filter out smart-diffs that exceed the LLM's maximum context window, as this tiny fraction of smart-diffs would incur substantial token costs to analyze across numerous context-window sized chunks.

In total, we find that just 2.6% of smart-diffs exceeded the model's maximum context window, and 71.7% of smart-diffs did not alter any package code or scripts, resulting in 7,661 smart-diffs that we apply LLM-based analysis to. For this experiment, we opt to utilize the GPT-4o model since it achieved the highest classification accuracy across our evaluation, as well as relatively faster execution times and cheaper token costs.

Flagged Package Updates: Our live analysis using GPT-4o flagged seven updates as malicious. We manually explore each of these updates and assess the LLM output to determine why they were flagged. Below, we describe each instance.

Two versions of the `react-native-bootsplash` package were flagged via the LLM due to significant alterations made to a javascript file containing obfuscated code. Manual investigation of this code revealed that it performs license validation to grant access to premium features of the package, which the package author sells. As such, the code obfuscation appears to be aimed at preventing users from

fraudulently bypassing license validation. We classify these instances as false positives of the 'sensitive operations' type.

The LLM also flagged one version of `cdk-assets`, as this update includes code which uploads the contents of a hard-coded file path to an Amazon S3 bucket. Looking deeper into this package revealed that this code is part of a test script that is not needed to run the package. We also classify this instance as a false positive of the 'sensitive operations' type.

Last, the LLM flagged four versions of the `ruhend-scraper` package, as the code for this package is entirely obfuscated and performs numerous network requests, which the LLM deemed highly suspicious. Manual investigation revealed this package to be a social media video scraper, which provides functionality for users to download videos from social media websites via network requests made to third-party services. Although we did not identify any explicitly malicious behaviors in this package, we note that unauthorized video scraping may be in violation of certain social media websites' terms of service, such as YouTube's [75], which the package provides functionality to scrape videos from, likely explaining the motivation for code obfuscation. Due to the suspicious and potentially terms of service violating nature of this package, we believe that flagging this package for manual analysis is beneficial from a security analyst's perspective.

Ultimately, we believe that this live experiment demonstrates smart-diff enabled LLM-based malware analysis can be practical at repository-scale, both in keeping pace with the growth of npm and in flagging only a manageable quantity of package updates.

VII. Conclusion

Malicious package updates pose a serious threat to the security of open-source software ecosystems and the software supply chains they enable. Motivated by limitations in existing approaches, our work provides a deeper investigation of the application of LLM-based security analysis within this problem space. We demonstrated that LLMs' first-order concerns, such as limited context and output classification, can be overcome to enable accurate and efficient detection of historical package malware at repository scale. However, we also uncovered deeper, second-order concerns regarding the reasoning capabilities of tested LLMs, giving rise to subtle forms of adversarial evasion that are capable of subverting LLM-based analysis. Ultimately, our findings demonstrate promising, yet nuanced potential for future work in employing LLMs, not as a replacement, but as a complement to traditional forms of security analysis.

VIII. Data Availability

We publicly release all code and data associated with this work to promote open science and replicability. We distribute these materials via the Open Science Framework [76], at the following link:

https://osf.io/92w4e/?view_only=c667861f653b44339893032fbe40e285

References

[1] OpenJS Foundation, "npm." https://www.npmjs.com/, July 2024.

[2] E. DeBill, "Modulecounts." http://www.modulecounts.com/, 2021.

[3] R. K. Vaidya, L. D. Carli, D. Davidson, and V. Rastogi, "Security issues in language-based sofware ecosystems," *CoRR*, vol. abs/1903.02613, 2019.

[4] C. Okafor, T. R. Schorlemmer, S. Torres-Arias, and J. C. Davis, "Sok: Analysis of software supply chain security by establishing secure design properties," in *SCORED*, 2022.

[5] Sonatype, "10th annualstate of the software supply chain." https://www.sonatype.com/state-of-the-software-supply-chain/introduction, 2024.

[6] D. Temple-Raston, "A 'worst nightmare' cyberattack: The untold story of the solarwinds hack." https://www.npr.org/2021/04/16/985439655/a-worst-nightmare-cyberattack-the-untold-story-of-the-solarwinds-hack, April 2021.

[7] thesamesam, "Faq on the xz-utils backdoor (cve-2024-3094)." https://gist.github.com/thesamesam/223949d5a074ebc3dce9ee78baad9e27, April 2024.

[8] R. J. Sofaer, Y. David, M. Kang, J. Yu, Y. Cao, J. Yang, and J. Nieh, "Rogueone: Detecting rogue updates via differential data-flow analysis using trust domains," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[9] F. N. Froh, M. F. Gobbi, and J. Kinder, "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*, SCORED '23, (New York, NY, USA), p. 41–49, Association for Computing Machinery, 2023.

[10] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *NDSS 2021*, Internet Society, 2021.

[11] A. Sejfia and M. Schäfer, "Practical automated detection of malicious npm packages," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (New York, NY, USA), p. 1681–1692, Association for Computing Machinery, 2022.

[12] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, (Red Hook, NY, USA), Curran Associates Inc., 2020.

[13] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: scaling language modeling with pathways," *J. Mach. Learn. Res.*, vol. 24, mar 2024.

[14] B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, E. T. Barr, and W. Le, "A comprehensive study of the capabilities of large language models for vulnerability detection," 2024.

[15] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large language model for vulnerability detection and repair: Literature review and the road ahead," 2024.

[16] M. Fu and C. Tantithamthavorn, "Linevul: a transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, (New York, NY, USA), p. 608–620, Association for Computing Machinery, 2022.

[17] M. D. Purba, A. Ghosh, B. Radford, and B. Chu, "Software vulnerability detection using large language models," pp. 112–119, 10 2023.

[18] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, "Transformer-based language models for software vulnerability detection," 2022.

[19] N. Zahan, P. Burckhardt, M. Lysenko, F. Aboukhadijeh, and L. Williams, "Shifting the lens: Detecting malware in npm ecosystem with large language models," 2024.

[20] D. Xue, G. Zhao, Z. Fan, W. Li, Y. Xu, Z. Liu, Y. Liu, and Z. Yuan, "Poster: An exploration of large language models in malicious source code detection," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, (New York, NY, USA), p. 4940–4942, Association for Computing Machinery, 2024.

[21] S. Shekhar, T. Dubey, K. Mukherjee, A. Saxena, A. Tyagi, and N. Kotla, "Towards optimizing the costs of llm usage," *arXiv preprint arXiv:2402.01742*, 2024.

[22] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," 2023.

[23] T. Li, G. Zhang, Q. D. Do, X. Yue, and W. Chen, "Long-context llms struggle with long in-context learning," 2024.

[24] A. Ouni, I. Saidani, E. Alomar, and M. W. Mkaouer, "An empirical study on continuous integration trends, topics and challenges in stack overflow," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, (New York, NY, USA), p. 141–151, Association for Computing Machinery, 2023.

[25] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *USENIX Security 19*, 2019.

[26] K. Chatzidimitriou, M. Papamichail, T. Diamantopoulos, M. Tsapanos, and A. Symeonidis, "Npm-miner: An infrastructure for measuring the quality of the npm registry," in *MSR 2018*, pp. 42–45, IEEE, 2018.

[27] B. Pfretzschner and L. ben Othmane, "Identification of dependency-based attacks on node.js," in *ARES*, 2017.

[28] S. Neupane, G. Holmes, E. Wyss, D. Davidson, and L. D. Carli, "Beyond typosquatting: An in-depth look at package confusion," in *32nd USENIX Security Symposium (USENIX Security 23)*, (Anaheim, CA), pp. 3439–3456, USENIX Association, Aug. 2023.

[29] R. Abdalkareem, V. Oda, S. Mujahid, and E. Shihab, "On the impact of using trivial packages: an empirical case study on npm and pypi," *Empirical Software Engineering*, vol. 25, 03 2020.

[30] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the diversity of software package popularity metrics: An empirical study of npm," in *SANER 2019*, 2019.

[31] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. Gonzalez-Barahona, "An empirical analysis of technical lag in npm package dependencies," 04 2018.

[32] T. Dey and A. Mockus, "Deriving a usage-independent software quality metric," *ESE*, vol. 25, 2020.

[33] S. Mujahid, R. Abdalkareem, and E. Shihab, "What are the characteristics of highly-selected packages? a case study on the npm ecosystem," 2022.

[34] E. Wyss, L. De Carli, and D. Davidson, "What the fork? finding hidden code clones in npm," in *ICSE 2022*, 2022.

[35] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the attack surface of node.js applications," in *RAID 2020*, USENIX Association, Oct. 2020.

[36] E. Wyss, A. Wittman, D. Davidson, and L. De Carli, "Wolf at the door: Preventing install-time attacks in npm with latch," in *ASIA CCS '22*, 2022.

[37] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem," *arXiv preprint arXiv:2201.03981*, 2022.

[38] A. Fass, M. Backes, and B. Stock, "Jstap: a static pre-filter for malicious javascript detection," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, (New York, NY, USA), p. 257–269, Association for Computing Machinery, 2019.

[39] M. Taylor, R. Vaidya, D. Davidson, L. De Carli, and V. Rastogi, "Defending against package typosquatting," in *NSS 2020*, 2020.

[40] F. Gauthier, B. Hassanshahi, and A. Jordan, "Affogato: Runtime detection of injection attacks for node.js," in *ISSTA/ECOOP 2018*, 2018.

[41] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?," in *ICSE-SEIP 2022*, 2022.

[42] M. Ohm, F. Boes, C. Bungartz, and M. Meier, "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*, ARES '22, (New York, NY, USA), Association for Computing Machinery, 2022.

[43] C. Huang, N. Wang, Z. Wang, S. Sun, L. Li, J. Chen, Q. Zhao, J. Han, Z. Yang, and L. Shi, "Donapi: Malicious npm packages detector using behavior sequence knowledge mapping," *arXiv preprint arXiv:2403.08334*, 2024.

[44] K. Garrett, G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, "Detecting suspicious package updates," in *ICSE-NIER 2019*, pp. 13–16, IEEE, 2019.

[45] M. Ohm and C. Stuke, "Sok: Practical detection of software supply chain attacks," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ARES '23, (New York, NY, USA), Association for Computing Machinery, 2023.

[46] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," 2024.

[47] Z. Zheng, K. Ning, J. Chen, Y. Wang, W. Chen, L. Guo, and W. Wang, "Towards an understanding of large language models in software engineering tasks," 2023.

[48] S. Kang, G. An, and S. Yoo, "A quantitative and qualitative evaluation of llm-based explainable fault localization," *Proceedings of the ACM on Software Engineering*, vol. 1, p. 1424–1446, July 2024.

[49] J. Wang, Z. Huang, H. Liu, N. Yang, and Y. Xiao, "Defecthunter: A novel llm-driven boosted-conformer-based code vulnerability detection mechanism," 2023.

[50] N. S. Mathews, Y. Brus, Y. Aafer, M. Nagappan, and S. McIntosh, "Llbezpeky: Leveraging large language models for vulnerability detection," 2024.

[51] Y. Yang, "Iot software vulnerability detection techniques through large language model," in *Formal Methods and Software Engineering* (Y. Li and S. Tahar, eds.), (Singapore), pp. 285–290, Springer Nature Singapore, 2023.

[52] L. Li and B. Gong, "Prompting large language models for malicious webpage detection," in *2023 IEEE 4th International Conference on Pattern Recognition and Machine Learning (PRML)*, pp. 393–400, 2023.

[53] M. A. Ferrag, M. Ndhlovu, N. Tihanyi, L. C. Cordeiro, M. Debbah, T. Lestable, and N. S. Thandi, "Revolutionizing cyber threat detection with large language models: A privacy-preserving bert-based lightweight model for iot/iiot devices," 2024.

[54] P. M. S. Sánchez, A. H. Celdrán, G. Bovet, and G. M. Pérez, "Transfer learning in pre-trained large language models for malware detection based on system calls," 2024.

[55] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," 2023.

[56] I. Shumailov, Z. Shumaylov, Y. Zhao, Y. Gal, N. Papernot, and R. Anderson, "The curse of recursion: Training on generated data makes models forget," 2024.

[57] N. Risse and M. Böhme, "Uncovering the limits of machine learning for automatic vulnerability detection," 2024.

[58] G. Dobocan, "State of npm 2023: The overview." https://blog.sandworm.dev/state-of-npm-2023-the-overview, Jul 2023.

[59] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2020.

[60] A. Sharma, "Popular 'coa' npm library hijacked to steal user passwords." https://www.bleepingcomputer.com/news/security/popular-coa-npm-library-hijacked-to-steal-user-passwords/, Nov 2021.

[61] @adam npm, "Reported malicious module: getcookies." https://blog.npmjs.org/post/173526807575/reported-malicious-module-getcookies.html, May 2018.

[62] G. A. Database, "Malicious package in sailclothjs." https://github.com/advisories/GHSA-m5pf-5894-jmx7, Sep 2020.

[63] StacklokLabs, "Jail." https://github.com/StacklokLabs/jail, Nov 2024.

[64] M. Bazon, "Uglyfijs." https://github.com/mishoo/UglifyJS, Feb 2017.

[65] j4k0xb, "Webcrack." https://github.com/j4k0xb/webcrack, Feb 2023.

[66] T. Dodona, "Dolos." https://dolos.ugent.be/, Mar 2022.

[67] P. Eggert, M. Haertel, D. Hayes, R. Stallman, and L. Tower, "Diff(1)." https://man7.org/linux/man-pages/man1/diff.1.html, Dec 2023.

[68] OpenAI, "Models." https://platform.openai.com/docs/models, May 2024.

[69] Google, "Gemini." https://gemini.google.com/app, February 2025.

[70] A. Kong, S. Zhao, H. Chen, Q. Li, Y. Qin, R. Sun, X. Zhou, E. Wang, and X. Dong, "Better zero-shot reasoning with role-play prompting," 2024.

[71] F. Hill, K. Cho, and A. Korhonen, "Learning distributed representations of sentences from unlabelled data," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (K. Knight, A. Nenkova, and O. Rambow, eds.), (San Diego, California), pp. 1367–1377, Association for Computational Linguistics, June 2016.

[72] T. Gao, X. Yao, and D. Chen, "Simcse: Simple contrastive learning of sentence embeddings," 2022.

[73] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, p. 321–357, June 2002.

[74] npm, "Public registry api." https://github.com/npm/registry/blob/main/docs/REGISTRY API.md, 2024.

[75] YouTube, "Terms of service." https://www.youtube.com/static?template=terms, Dec 2023.

[76] "Open science framework." https://osf.io, Jan 2022.