Work-in-progress: Uncovering the Invisible: A Large-Scale Analysis of Service Worker Security Risks in the Wild

Skanda Dhanushkanda, Mustafa Ibrahim, and Shuai Hao
Old Dominion University, Norfolk, VA, USA
{sdhan005, mibra004, shao}@odu.edu

***Abstract*—Service Workers (SWs) are a cornerstone of modern Progressive Web Applications, enabling native-like functionalities such as offline access, caching, and background synchronization. However, these capabilities also broaden the attack surface, as SWs persist in the user's browser across sessions and can be leveraged to inject malicious content over time. In particular, adversaries can exploit certain sensitive functions, which refer to standard or custom JavaScript functions and APIs that become security-critical when used with untrusted or unsanitized input, to achieve more subtle and stealthy attacks for end-users. In this study, we conduct a large-scale measurement study of security risks in SWs across the web, analyzing ≈12K scripts collected from the top 1M websites. We show that traditional static analysis tools are largely ineffective at detecting vulnerabilities involving sensitive functions due to their static, rule-based nature. To address this limitation, we develop a systematic, LLM-assisted vulnerability analysis framework to detect vulnerabilities arising from sensitive functions in SWs at scale. Our results show that 80% of analyzed SWs contain potentially vulnerable uses of sensitive functions, with many previously overlooked by existing static analysis tools. These findings expose systemic weaknesses in the SW ecosystem and demonstrate how LLM-assisted analysis offers a promising path for scalable, semantic vulnerability assessment of modern web applications beyond traditional methods.**

I. Introduction

Progressive Web Applications (PWAs) have emerged as a paradigm to bridge the gap between native and web applications. At its core is *Service Worker* (SW), a JavaScript program that is executed in the background and provides extensive features such as caching, offline access, push notifications, and background synchronization. However, these elevated privileges also make SWs a high-value target for miscreants. If compromised, a malicious SW can intercept network traffic, manipulate responses, and undermine the overall security of web applications. Furthermore, since SWs can serve cached content independently of network connections, malicious payloads injected through SWs may persist and continue operating even after the initial vulnerability has been fixed or the network connectivity is restored, enabling stealthy, long-lived attacks.

Static analysis tools have been primarily used to achieve rule-based scanning for identifying known security patterns in JavaScript. While these tools can be used to analyze SW scripts, none of them provides SW-specific rules for detecting vulnerabilities tied to SW functionalities and operations, resulting in ineffectiveness in capturing SW-specific security risks. When it comes to handling contextual or semantic vulnerabilities, such as distinguishing between safe and unsafe uses of the fetch API or reasoning about the implications of dynamic code loading and inter-thread message passing, these tools often fall short despite their strengths in identifying syntactic or structural issues. To this end, JavaScript's inherent dynamism and its asynchronous, event-driven execution model make static analysis alone inadequate for uncovering subtle logic flaws or identifying malicious intent [1], [2], [3].

To overcome these limitations, we explore leveraging Large Language Models (LLMs) as the primary mechanism for analyzing SW vulnerabilities. Unlike rule-based static analysis tools, LLMs can perform semantic reasoning and recognize higher-level behavioral patterns, enabling them to identify subtle logic flaws, insecure design decisions, or suspicious interactions that static techniques cannot capture. Recent work has shown that LLMs can move beyond syntactic pattern matching to provide contextual understanding of program behavior [1]. Leveraging these capabilities, our approach focuses on evaluating the behavioral semantics of SW code, allowing the detection of potentially compromised logic or misuse of SW APIs that traditional static analysis is unable to uncover.

To evaluate real-world security implications of SWs and sensitive functions, we employ three state-of-the-art LLMs to analyze SW scripts at scale. Our analysis detects generic vulnerabilities, checks if they stem from misuse of sensitive functions, and evaluates whether those functions are susceptible to exploitation. Across our dataset of SWs collected from Tranco's top 1M websites [4], we discover ≈200 unique websites exhibiting high-severity vulnerabilities that are potentially exploitable. Given the increasing adoption of SWs, our findings highlight an emerging and largely overlooked attack surface. Our contributions are summarized as follows:

- We present a large-scale, systematic study to leverage LLMs to identify security issues in SWs deployed in the wild.
- We propose a multi-phase framework that enables semantic vulnerability analysis of SW scripts, uncovering widespread yet overlooked misuse of sensitive functions.

Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-06-0
https://dx.doi.org/10.14722/madweb.2026.23040
www.ndss-symposium.org

- We conduct a large-scale empirical study of over 6,400 real-world SW scripts, categorizing their vulnerability patterns and exploring their exploitation potential.

II. Background and Related Work

A. Background

Progressive Web Applications. A Progressive Web Application (PWA) [5] is a program that is built for web services but provides experience and features like a native application running on particular platforms using specific programming languages, *e.g.*, Java for Android platform and Objective-C or Swift for iOS. Meanwhile, Service Workers (SWs) [6], as a core component of PWAs, are also widely deployed by web services to enable key functionalities and operations of PWAs, such as offline usage, push notifications, background synchronization, and effective caching.

Service Workers. A SW script is an event-driven JavaScript program that runs independently of web pages and acts as an intermediary proxy between the client's browser and the web server, intercepting and handling network packets. When a user accesses a PWA-enabled web service, the SW is downloaded and registered in the user's browser, with a scope indicating the URL path under which the SW can operate web pages. After registration, the SW is installed by initializing the browser's cache with the resources designated in the script. Once activated, the SW can handle function events such as fetch, push, and sync, enabling capabilities like background execution and push notifications. On the other hand, unlike ordinary JavaScript, SWs persist across browsing sessions and remain active even after the user navigates away from the page. In addition, SWs have privileged access to browser storage and caching mechanisms, allowing them to access potentially sensitive resources. These capabilities make SW scripts a high-value target for adversaries and could have significant security implications for the web ecosystem.

Sensitive Functions. *Sensitive functions* generally refer to a class of JavaScript functions that process user input or enable dynamic execution. These functions are commonly used to support content rendering, real-time updates and personalized user experience. However, when invoked on unsanitized or malformed data, they may introduce significant security risks, making their use within SWs particularly critical to examine.

Sensitive function was first introduced by Chinprutthiwong *et al.* [7] when studying the misuse of standard JavaScript functions such as importScripts and eval. Beyond these built-in APIs, developers often define custom functions for content generation or dynamic execution. Accordingly, in this paper, we examine both standard and custom JavaScript functions, as both can enable code injection or unintended runtime modification when improperly paired with user-controlled input.

B. Related Work

JavaScript is a core component in modern web architecture, which introduces a variety of security vulnerabilities. Previous research has studied how insecure practices of using JavaScript on the web introduces threat such as [8], [9], [10], [11].

As a specific JavaScript program, SWs have also been widely examined for their misuse across a variety of attack scenarios. Jeong *et al.* [12] presented a comprehensive taxonomy of these vulnerabilities, while Subramani *et al.* [13] conducted a Systematization of Knowledge (SoK) that not only surveys existing threats but also identifies previously unexplored attack surfaces. More specifically, Chinprutthiwong *et al.* [7] showed that attackers can inject malicious code into benign SWs, leading to persistent Cross-Site Scripting (XSS) attacks. Later, Chinprutthiwong *et al.* [14] and Squarcina *et al.* [15] explored vulnerabilities arising from the interaction between SWs and the DOM. Watanabe *et al.* [16] further demonstrated that SWs can facilitate persistent Man-in-the-Middle (MitM) attacks by intercepting and modifying network traffic.

Prior efforts have also explored how SWs can be exploited for privacy infringements. For example, Lee *et al.* [17] and Karami *et al.* [18] leveraged SWs to infer user's browser history and visited websites. Van Goethem *et al.* [19], [20] explored side-channel attacks of SWs, revealing that timing attacks against SW cache could leak private user behaviors.

Additionally, SW misuse also involves resource abuse through background processing capabilities. Papadopoulos *et al.* [21] demonstrated the feasibility of installing malicious SWs to construct web-based botnets for DDoS attacks. Beyond this, Lee *et al.* [17] and Subramani *et al.* [22] showed how attackers could exploit Push Notifications to deliver malvertising content or lure victims into phishing campaigns.

III. Methodology

In this section, we outline our methodology for the security analysis of SW scripts. We begin with data collection and preliminary analysis showing the ineffectiveness of traditional static analysis, and then present our LLM-assisted approach for in-depth vulnerability detection and exploitation analysis.

Overview. We first collect unique SW scripts through a crawl of the Tranco top 1M websites [4]. We begin by using popular static analysis tools to assess their effectiveness in identifying vulnerabilities involving SWs and sensitive functions. To overcome the limitations of static tools, we explore an LLM-assisted approach that can provide deeper semantic reasoning and behavioral insights of JavaScript code in SWs.

Figure 1 illustrates the framework of our analysis workflow, which implements a two-phase analysis pipeline based on LLMs: ❶ General Vulnerability Detection and ❷ Vulnerability Detection by Sensitive Function Misuses. The first phase focuses on the broad identification of all vulnerability types. In the second phase, we filter the *Phase I* results to pinpoint vulnerabilities caused by the misuse of sensitive functions. The results from *Phase II* are then subjected to cross-model comparison and clustering into five coherent groups, concluding with an exploitation analysis, which assesses the real-time feasibility of the identified sensitive function misuses.

A. Data Collection

To collect a comprehensive SW dataset, we develop a custom Python script using Puppeteer [23] to access each domain

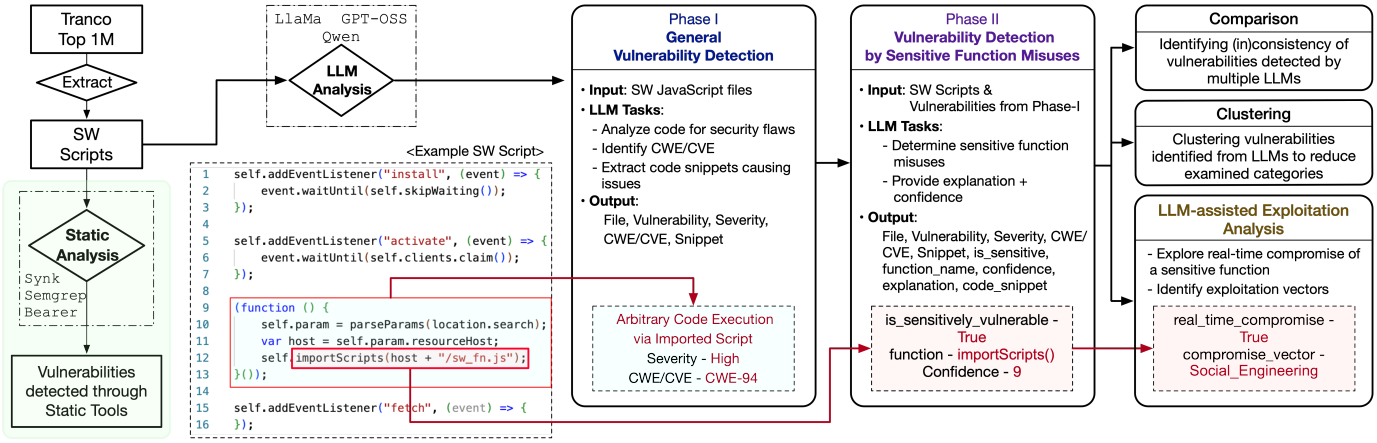

Fig. 1: Overview of Our Analytic Framework

TABLE I: Results of Static Analysis for Collected SW Scripts

Tools	Scripts	Rows	CWE
Snyk [24]	2,357	3,688	10
Semgrep [25]	1,768	7,187	13
Bearer [26]	4,115	21,633	13

in Tranco's top list. Our crawler verifies the availability of serviceWorker API for each domain and extracts all active registrations via navigator.serviceWorker.getRegistrations(), including their script URLs and source code.

We observe that many websites deploy identical SW scripts, *e.g.*, often due to cross-regional deployments belonging to the same organization or the adoption of platforms enabling shared templates (*e.g.*, WordPress and Wix.com). Consequently, we filter out duplicated scripts and retain only unique scripts for further analysis, resulting in 6,437 (51.1%) unique scripts.

B. Preliminary Study: Static Analysis of SW Scripts

Traditional static analysis inspects a program's source code or bytecode without executing it in order to identify potential security vulnerabilities [27]. To assess the security posture of the SW scripts collected in our dataset, we applied several widely used static analysis tools, including Semgrep, Snyk, and Bearer. Table I presents the results of static analysis tools.

We can see that each of the tools flags a narrow set of vulnerabilities due to their limited scope of rule sets. For instance, Semgrep primarily detects "Inefficient RegEx complexity" (71.4%) and Bearer detects "Leakage of information in logger message" (73%). Both of these tools also flag hard-coded secrets or API keys. Even though these do not fall into sensitive function-related threats, Snyk can detect "Insufficient validation" in postMessage() in around 1,891 SW scripts. We also observe that Semgrep flags certain uses of eval() as injection threats, and RegExp() leading to potential ReDoS.

However, through manual inspection, we also identify that these tools still struggle to detect misuse of many sensitive functions such as importScripts() and fetch(). This limitation mainly stems from the fact that these vulnerabilities emerge only when supplied with unsanitized input, which typically requires data-flow analysis so as to make traditional static analysis inadequate for SWs and sensitive functions.

C. LLMs-Assisted Vulnerability Detection

LLMs have recently shown strong potential for code analysis and vulnerability detection [1], [28], [29], [30], [31]. Motivated by the limitations of static analysis in capturing SW-specific semantic risks, we employ LLMs to uncover vulnerability patterns that rule-based tools frequently miss.

1) LLMs: We evaluate three open-source LLMs: **Llama-4-Maverick-17B** [32], **Qwen-32B** [33], and **gpt-oss-120b** [34]. These models were selected for their popularity and availability in our computing environment. All models receive the same prompts and SW code. Qwen's 32k-token context window is the smallest among them, and since our prompt consumes roughly 2.8k tokens, scripts exceeding the remaining budget are split into 30k-token chunks. The identical chunking strategy is applied to all models, ensuring uniform input despite Llama and GPT having larger context windows.

2) Prompt Design: To achieve effective code analysis for SW scripts using LLMs, we employ a structured prompting strategy that guides the model through hierarchical reasoning.

We developed a two-phase prompting scheme that first identifies potential vulnerabilities (*Phase I*) and then determines which of them stem from the misuse of sensitive functions (*Phase II*). By decoupling these tasks, the model can engage in more focused and distinct reasoning, avoiding casual analysis that often leads to degraded performance and LLM hallucinations. This strategy also mirrors standard security auditing workflow, where an initial survey of issues is preceded by an in-depth analysis. Moreover, this facilitates our assessment regarding the effectiveness of LLMs in identifying security issues of SWs' JavaScript code, particularly those related to the misuse of sensitive functions.

Phase-I Prompt. We first prompt the LLMs to act as a researcher with expertise in web security to review the provided code for identifying potential vulnerabilities. The models produce output with a formatted schema that includes the vulnerability and its description, severity (Low/Medium/High), CWE/CVE, and the code snippet from which the vulnerability originates. This ensures consistent results and allows seamless integration into our large-scale analysis pipeline.

We explicitly instruct the model to assign a CWE or CVE only when a clear and justifiable mapping exists; otherwise, the field remains empty. We explore both CWEs, which describe classes of weaknesses, and CVEs, which represent specific vulnerability instances, to evaluate whether LLMs can map findings to broader vulnerability categories or concrete instances, respectively. This is because we observe that a substantial portion of the high-severity findings (approximately 28%) involve previously undocumented patterns of sensitive function misuse in SWs. By retaining these cases and categorizing them as "Unlabeled" in Phase I, we capture novel vulnerability classes that traditional static analyzers and existing taxonomies fail to detect, many of which later emerge as the dominant clusters in Section III-D.

Phase-II Prompt. Next, we prompt the LLM as a cybersecurity expert specializing in JavaScript to detect vulnerabilities caused by sensitive function misuse. The definition of sensitive functions is given as "a JavaScript function or API (either Standard or Custom) that introduces security vulnerabilities when used with untrusted or unsanitized input." Moreover, to help the LLM understand the context of their misuse, we provide an example for the misuse (importScripts()), demonstrating how the unvalidated data flowing into a sensitive function can introduce risks. As shown in Figure 1, the SW extracts a user-controlled parameter through location.search and concatenates it into the URL used by importScripts() allowing an attacker controlled script to be execured within the scope of the SW. The results obtained from Phase-I are used as input to Phase-II. The prompt asks the LLM to identify cases where sensitive functions receive unsanitized dynamic input producing a formatted output shown in Figure 1. The prompts used in our study are shown in Appendix B.

D. Clustering Vulnerabilities

LLMs produce a diverse set of vulnerabilities, as seen in Tables II and III. To organize and further interpret these results, we then cluster the identified vulnerabilities from Phase-II based on their severity levels.

Initially, we use text vectorization to transform each vulnerability description into a high-dimensional sparse form through the TF-IDF (Term Frequency-Inverse Document Frequency) approach. This method highlights the relative significance of key terms while reducing the impact of common words. Subsequently, we apply the K-Means clustering algorithm (with $K \in \{5, 6, 7, 10\}$) to classify semantically similar vulnerabilities in the TF-IDF feature space. The resulting clusters are then processed by LLMs to create concise titles and summaries for each cluster to improve interpretability. Through empirical evaluation, we determine that five clusters per severity level can achieve the most coherent grouping, balancing semantic distinctness and intra-cluster consistency.

E. Exploitation Analysis

To validate whether the identified vulnerabilities can be exploited in the real world, we perform an extensive analysis to deduce the attack preconditions and the practical feasibility of triggering each vulnerability (prompt in Appendix B). The LLMs are instructed to provide two new columns for each case of a sensitive function misuse from Phase-II: attack-vector, which clearly states the real-world technique an attacker could use (*e.g.*, Crafted URL inputs, compromised push payload), and real-world-feasibility (High/Medium/Low), which estimates the likeliness for an attacker to satisfy the preconditions in realistic deployments. This classification prioritizes the vulnerabilities with high practical feasibility

TABLE II: Detected vulnerabilities across LLMs (Phase-I).

Metrics		Llama	GPT	Qwen
⟨**Script, Vulnerability**⟩		22,160	15,024	9,950
Severity	High	6,276	5,683	5,011
	Medium	11,932	7,164	4,442
	Low	3,929	2,170	443
Vulnerability Mapping	CWE	18,810 (*204*)	14,469 (*143*)	8,605 (*216*)
	CVE	211 (*23*)	19 (*4*)	14 (*10*)
	Unlabeled	3,139	422	1,553
Detected SW Scripts		5,813 (90.3%)	5,445 (84.6%)	5,211 (81%)
Vulnerabilities		6,043	12,260	5,617

IV. Results & Analysis

In this section, we present our findings obtained through our LLM-assisted detection schema described in Section III-C.

A. Results of LLM-Based Analysis

Using our dataset of 6,437 unique SW scripts, we created separate corpora using the three LLM models for vulnerabilities detected through Phase-I, and for those that were caused by the misuse of sensitive functions through Phase-II. In our dataset, each row corresponds to a specific SW script and a vulnerability detected (<*script, vulnerability*>), therefore a script may appear in multiple rows if multiple vulnerabilities are identified. Hereafter, we refer to this as an <*s,v*> pair.

1) Phase-I: General vulnerability detection: Table II summarizes the vulnerability detection capabilities of the evaluated LLMs. Overall, Llama demonstrates the highest detection and coverage, generating the largest dataset (22,160 <*s,v*> pairs) and flagging potential vulnerabilities in more than 90% of the unique scripts (5,813 out of 6,437). GPT and Qwen follow with detection rates of 85% (5,445 scripts) and 79% (5,094 scripts), respectively. The overlap analysis gave us 4,957 scripts detected with at least one vulnerability by all 3 LLMs.

We used our LLMs to assign each <*s,v*> pair a *Severity index*, categorized as High, Medium, or Low. LLaMA flags the most `High` severity findings (6,292 out of 22,177 <*s,v*> pairs) in 4,380 scripts. It also identifies Medium and Low severity issues on 4,847 and 3,151 scripts, respectively. GPT exhibits a comparable pattern, with 5,683 High severity <*s,v*> rows (4,426 scripts), and Medium and Low severity issues on 4,353 and 1,780 scripts. Qwen is the most conservative, assigning high severity to 5,011 <*s,v*> rows (3,855 scripts) and detecting far fewer low-severity issues in 389 scripts.

In terms of mapping the detected vulnerabilities to standardized CWE/CVE identifiers, we observed more than 200 unique CVE/CWEs for both Llama and Qwen, but they also couldn't map around 15% of <*s,v*> rows in their respective

TABLE III: Sensitive function misuse identified in Phase-II.

Metrics		Llama	gpt	Qwen
Unique SW Scripts		4,635	4,926	2,089
Vulnerabilities		3,437	8,400	1,948
True Sensitive		11,822	10,192	3,342
Severity	• High	5,506	4,510	2,197
	• Medium	9,684	4,741	1,111
	• Low	1,650	937	33
CVE		10	3	0
CWE		122	118	113
Unlabeled		1,785	422	1,031
Standard		24	6	11
Custom		204	177	216

datasets. GPT demonstrates better classification consistency, identifying 147 unique CVE/CWEs identifiers, but with less than 3% unlabeled <*s,v*> rows. An overlap analysis of the combined CWE/CVE mappings across all models yields 422 unique values, among which 21 are tagged under MITRE's 2025's Top 25 Most Dangerous Software Weaknesses [35].

2) Phase-II: Identify sensitive function misuse: Next, we evaluate whether the vulnerabilities identified in Phase-I involve sensitive function misuse. In this phase, each row from Phase-I is processed independently, and the models reassess it to determine whether sensitive functions are involved. Specifically, the LLMs extract the function responsible for the potential misuse and classify it as either a *standard* library function or a *custom* implementation.

Table III summarizes the Phase-II results. LLaMA shows the highest coverage, identifying true sensitive function misuse in 4,635 unique scripts (72% of the dataset). In total, the model verifies misuse in 11,822 of the 22,177 rows from Phase-I. GPT follows, detecting misuse in 4,926 unique scripts and verifying 10,192 out of 15,023 processed rows. Qwen is the most conservative, flagging 2,089 unique scripts and confirming misuse in 3,342 of 9,950 processed rows.

A key finding is the distinction between function types. Across all three models, vulnerabilities are overwhelmingly attributed to custom, user-defined functions, indicating that the primary security risk lies in proprietary application logic rather than the misuse of standard APIs. For example, Llama identifies 204 unique custom functions compared to only 24 unique standard library functions, gpt identifies 6 standard and 177 custom functions, and Qwen follows a similar trend (216 custom vs 11 standard). This strong bias toward custom function findings underscores the necessity of deep semantic analysis provided by LLMs, as traditional static analysis often struggles to trace control and data flow through user-defined code structures. We show a comparison of detected vulnerable scripts across the LLMs in Appendix C.

3) Results of exploitation analysis: We conduct an exploitation analysis to examine the exploitability of identified vulnerabilities (§III-E). As shown in Table IV, Qwen identifies a total of 5,366 rows across 2,699 unique SW scripts. Among these, 817 scripts are classified as potentially compromisable under our threat model. These scripts can be exploited through various attack vectors, including social engineering, XSS, server-side injection, or their combinations.

TABLE IV: Real-time Compromise Analysis using Qwen.

Attack Vectors	# Unique Scripts
Social Engineering	349
Vector Combinations	306
Server-Side Injection	79
XSS	29
CSRF	54
Total	**817**

TABLE V: Common Vulnerability Clusters Across LLMs

Unified Cluster Theme	LLama	GPT	Qwen
Fetch Handler Exploitation	1	2	5
Cache Integrity Compromise	2	1	3
Notification Payload URL Injection	3,4,5	5	1,2
Untrusted Script Loading via `importScripts()`	—	4	4
Message-Driven Attacks	—	3	—

B. Clustering

Phase II produces numerous high-severity vulnerability reports across the three evaluated LLMs. Through this, we identify five dominant and recurring vulnerability classes when analyzing SW scripts in the Tranco Top-1M dataset. Across roughly 12,000 scripts, more than 80% exhibit at least one flaw involving the use of untrusted input in sensitive SW functions.

Table V lists the top clustered vulnerability categories and their rankings in each LLM model. We can see that each model emphasizes different symptoms, *e.g.*, LLaMA and Qwen frequently identify arbitrary code execution through unvalidated event data, while GPT highlights unsafe cache serving paths.

We employ our LLM-refined k-means clustering (§III-D) to merge hundreds of model-specific findings into unified vulnerability classes. For example, the largest cluster, "SW Notification Payload Injection," aggregates over 180 findings ranging from "XSS via unsanitized push notification data" and "phishing via untrusted notification content" to "unvalidated URL redirects in notification actions." Although these findings span multiple CWE categories and differ in phrasing or perceived severity, they all originate from the same underlying behavior: SW scripts directly forwarding untrusted fields from push payloads into DOM-sink APIs such as showNotification, clients.openWindow, and postMessage without sanitization or origin verification. This pattern repeats across the remaining clusters and demonstrates that LLM-based semantic clustering effectively consolidates heterogeneous model outputs into coherent root-cause vulnerability classes. These results highlight a significant advantage over traditional static analysis, which often fails to capture such behavioral patterns.

C. Validation of Detected Vulnerabilities

We next illustrate how sensitive functions can be misused with unsanitized input and how such vulnerabilities can be exploited in real-world scenarios.

Arbitrary Remote Code Execution. The importScripts API is widely used to import scripts into SW's scope. From our curated dataset, we find that almost 45% of SWs rely on this API, and the script paths are statically coded (*i.e.*, a fixed literal string), dynamically constructed (URLs built at runtime using

variables, parameters, *etc.*), or a hybrid of both. Although static resources may still introduce security issues depending on their content, we exclude them from our analysis as they cannot be manipulated by a malicious actor.

```
1 !function () {
2   "use strict";
3   self.params = location.search.substring(1).split("&")
    ↪  .reduce(function (s, r) {
4     var t = r.split("="),
5       e = t[0],
6       a = t[1];
7     return s[e] = a, s;
8   }, {});
9   var s = self.params.resourcesHost ||
    ↪  "https://website1.com";
10  self.importScripts(s +
    ↪  "/statics/website1/sw-router.js");
11 }();
```

Fig. 2: SW importing a script using importScripts().

Figure 2 shows a vulnerable SW script from a real website (illustrated as website1.com). A malicious actor can craft URLs with the domain name of the legitimate website and add a query parameter pointing to resources of their choosing. This crafted URL can be delivered to victims using typical social engineering techniques, such as phishing emails or links embedded in third-party websites. Once the user is lured to click on this link, the SW extracts the query parameters using the location.search without any validation or sanitization (line 3) and splits the key, value pairs, extracting and storing them in an object 's' (lines 4-8). Finally, as shown in lines 9-11, the resourceHost value is directly appended inside the importScripts() call, allowing the SW to execute this script.

```
1 const redirect = location => new Response(null, {
2   status: 302,
3   headers: { 'Location': location }
4 });
5 async function applyUserParams(db, url) {
6   return redirect(decodeURIComponent(url.href));}
```

Fig. 3: SW using a custom function for redirection.

Open Redirect via Unvalidated Parameters. In addition to standard APIs in JavaScript, custom functions can also be sensitive. One such case we observed is shown in Figure 3, where a user-crafted URL (Line 6) can be directly passed into a user-defined function redirect() (Lines 1-4). The attacker crafts a URL under a legitimate domain, augments it with malicious query parameters, and distributes it through social engineering such as phishing emails. When a victim follows this link, the unvalidated URL is propagated to the redirect logic, enabling the attacker to inject an external destination. As a result, the SW's intended protection against redirection is bypassed, enabling arbitrary redirects controlled by the attacker. This issue could be addressed by validating user-derived inputs, preventing untrusted or user-controlled query parameters from influencing redirect behavior, and constraining redirects strictly to predefined, trusted endpoints.

V. Discussion

A. Limitations

Despite the effectiveness of our approach, several limitations remain. First, the absence of a standardized ground-truth dataset for JavaScript vulnerabilities prevents rigorous quantitative evaluation. Consequently, we rely on indirect validation strategies, including cross-model consistency analysis and expert manual inspection.

Second, we observe that distinct LLMs often recognize the same vulnerability while assigning different CWE/CVE labels, likely due to inconsistencies in how these models map vulnerability patterns to diverse taxonomies, thereby limiting the reliability of automated triaging and straightforward cross-model comparison.

Finally, our collected dataset may include both obfuscated and minified SW scripts. While manual inspection did not identify explicitly obfuscated code samples, there may still exist subtle or partially obfuscated cases that are difficult to detect without systematic analysis.

B. Future Work

Our initial assessment of vulnerabilities in SW scripts largely relies on static tools, as the dynamic analysis remains infeasible in our LLM-based analysis due to the requirement for a more complex runtime environment such as browser instrumentation and event-driven behavior tracing. This may limit our observation for runtime behavior such as DOM modifications and asynchronous logic. In the future, we plan to develop enhanced framework to further assess such vulnerabilities in various dynamic tools such as Foxhound [36]. To further analyze the effectiveness of LLMs in detecting vulnerabilities and minimize hallucinations, we will also expand our analysis to additional LLMs with varying architectures, training data, and reasoning capabilities.

Moreover, we plan to develop a framework to enable *Proofs-of-Concept* for each detected threat, and evaluate its effectiveness by re-establishing our experimental environment. To achieve this, we will develop both a client and an attacker server, and induce the client to interact with the malicious input or link to evaluate if the reported threat can be realistically exploited in practice.

Also, we will leverage vulnerabilities identified in prior SW security studies and real-world disclosures as ground truth, evaluating whether LLMs can consistently rediscover these known issues. This will allow us to assess LLM performance beyond synthetic or manually curated examples.

Prior studies have evaluated LLM-based vulnerability detection using datasets for languages such as C/C++ and Python, but no comparable benchmark exists for JavaScript and Service Workers. As future work, we aim to construct a standardized SW vulnerability dataset by aggregating and validating findings from our large-scale analysis, enabling large-scale evaluation for LLM-assisted approaches for security analysis.

VI. Conclusion

In this paper, we present an LLM-assisted, multi-phase framework for analyzing security vulnerabilities in Service Workers (SWs). Through a large-scale study of over 6,400 distinct SW scripts, we find that approximately 80% of them use sensitive features in potentially unsafe ways. Our exploitation analysis further validates that many of these vulnerabilities are practically exploitable. Future work will focus on extending this approach to additional LLMs and developing a standardized, ground-truth dataset for JavaScript SW vulnerabilities.

REFERENCES

[1] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, "LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights," *ACM Computing Surveys*, 2025.

[2] A. Walker, M. Coffey, P. Tisnovsky, and T. Cerny, "On Limitations of Modern Static Analysis Tools," in *Information Science and Applications*, 2019.

[3] A. W. Marashdih, Z. F. Zaaba, and S. M. Almufti, "The Problems and Challenges of Infeasible Paths in Static Analysis," *International Journal of Engineering & Technology*, 2018.

[4] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczynski, and W. Joosen, "Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[5] MDN Web Docs, "Progressive Web Apps," https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps, 2025.

[6] ——, "Service Worker API," https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API, 2025.

[7] P. Chinprutthiwong, R. Vardhan, G. Yang, and G. Gu, "Security Study of Service Worker Cross-Site Scripting," in *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, 2020.

[8] C. Yue and H. Wang, "Characterizing Insecure JavaScript Practices on the Web," in *Proceedings of the 18th International Conference on World Wide Web*, 2009.

[9] M. Kluban, M. Mannan, and A. Youssef, "On Measuring Vulnerable Javascript Functions in the Wild," in *Proceedings of the ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*, 2022.

[10] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the World Wide Web from Vulnerable JavaScript," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2011.

[11] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra, "Automated Analysis of Security-Critical Javascript APIs," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[12] Y. Jeong and J. Hur, "A survey on vulnerabilities of service workers," in *Proceedings of the 13th International Conference on Information and Communication Technology Convergence*, 2022.

[13] K. Subramani, J. Jueckstock, A. Kapravelos, and R. Perdisci, "SoK: Workerounds - Categorizing Service Worker Attacks and Mitigations," in *Proceedings of the 7th IEEE European Symposium on Security and Privacy*, 2022.

[14] P. Chinprutthiwong, R. Vardhan, G. Yang, Y. Zhang, and G. Gu, "The Service Worker Hiding in Your Browser: The Next Web Attack Target?" in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021.

[15] M. Squarcina, S. Calzavara, and M. Maffei, "The Remote on the Local: Exacerbating Web Attacks Via Service Workers Caches," in *Proceedings of the IEEE Security and Privacy Workshops*, 2021.

[16] T. Watanabe, E. Shioji, M. Akiyama, and T. Mori, "Melting Pot of Origins: Compromising the Intermediary Web Services that Rehost Websites," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.

[17] J. Lee, H. Kim, J. Park, I. Shin, and S. Son, "Pride and Prejudice in Progressive Web Apps: Abusing Native App-like Features in Web Applications," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[18] S. Karami, P. Ilia, and J. Polakis, "Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2021.

[19] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The Clock is Still Ticking: Timing Attacks in the Modern Web," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[20] T. Van Goethem, M. Vanhoef, F. Piessens, and W. Joosen, "Request and Conquer: Exposing Cross-Origin Resource Size," in *Proceedings of the 25th USENIX Security Symposium*, 2016.

[21] P. Papadopoulos, P. Ilia, M. Polychronakis, E. P. Markatos, S. Ioannidis, and G. Vasiliadis, "Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[22] K. Subramani, X. Yuan, O. Setayeshfar, P. Vadrevu, K. H. Lee, and R. Perdisci, "When Push Comes to Ads: Measuring the Rise of (Malicious) Push Advertising," in *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2020.

[23] Chrome DevTools, "Puppeteer," https://pptr.dev/, 2024.

[24] Snyk, https://security.snyk.io.

[25] Semgrep, https://github.com/semgrep/semgrep.

[26] Bearer, https://github.com/Bearer/bearer.

[27] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using Static Analysis to Find Bugs," *IEEE Software*, 2008.

[28] C. Fang, N. Miao, S. Srivastav, J. Liu, R. Zhang, R. Fang, R. Tsang, N. Nazari, H. Wang, H. Homayoun *et al.*, "Large Language Models for Code Analysis: Do LLMs Really Do Their Job?" in *Proceedings of the 33rd USENIX Security Symposium*, 2024.

[29] H. Li and L. Shan, "LLM-based Vulnerability Detection," in *International Conference on Human-Centered Cognitive Systems*, 2023.

[30] Y. Li, P. Branco, A. M. Hoole, M. Marwah, H. M. Koduvely, G.-V. Jourdan, and S. Jou, "SV-TrustEval-C: Evaluating Structure and Semantic Reasoning in Large Language Models for Source Code Vulnerability Analysis," in *IEEE Symposium on Security and Privacy*, 2025.

[31] Z. Sheng, F. Wu, X. Zuo, C. Li, Y. Qiao, and L. Hang, "LProtector: An LLM-driven Vulnerability Detection System," *IEEE 4th International Conference on Data Science and Computer Application*, 2024.

[32] Meta, "Llama-4-maverick-17b-128e-instruct," https://huggingface.co/meta-llama/Llama-4-Maverick-17B-128E-Instruct, 2025.

[33] Alibaba Cloud, "Qwen3-32B," https://huggingface.co/Qwen/Qwen3-32B, 2025.

[34] OpenAI, "gpt-oss-120b," https://huggingface.co/openai/gpt-oss-120b, 2025.

[35] The MITRE Corporation, "2025 CWE Top 25 Most Dangerous Software Weaknesses," https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html, 2025.

[36] S. Calzavara, S. Casarin, and R. Focardi, "Dynamic Security Analysis of JavaScript: Are We There Yet?" in *Proceedings of the ACM Web Conference*, 2025.

APPENDIX

A. Ethics

Our study aims to identify potential vulnerabilities present in Service Worker scripts deployed by live websites in the wild, and does not target any specific victim web services or real users. We collect and analyze only publicly accessible resources and do not interact with, modify, or disrupt the normal operation of any website. Therefore, we consider the ethic risks of such investigation negligible relative to the broader positive impact of this work.

B. Prompts of LLM-assisted Vulnerability Detection

1) Phase-I prompt used to detect all vulnerabilities in a given SW script:

```
system_prompt = """
You are a PhD researcher specializing in web security, with a
↪  specific focus on discovering novel and known vulnerabilities
↪  in JavaScript Service Workers.
Your expertise includes identifying issues like insecure caching
↪  strategies, scope creep, race conditions in event handling, and
↪  improper cross-origin resource sharing within worker contexts.

CRITICAL OUTPUT RULES:
1. Output ONLY valid CSV data rows based on your rigorous analysis.
2. Do NOT include a CSV header row.
3. Do NOT include markdown backticks (```csv or ```).
4. Do NOT include ANY conversational text, introductions, or
↪  conclusions.
5. If no vulnerabilities are found, return an empty string.
6. Columns must be strictly: File Name, Vulnerability, Severity
↪  (Low/Medium/High), CWE/CVE, Code Snippet
"""
```

2) Phase-II prompt used to detect vulnerabilities caused by the misuse of sensitive JavaScript functions:

```
system_prompt = """
You are a cybersecurity expert specializing in JavaScript security
↪  and the detection of vulnerabilities caused by sensitive
↪  function misuse.
```

```
A sensitive function in a script is a JavaScript function or API
↪  (both standard or custom) that can introduce security risks
↪  when used with untrusted or unsanitized input.
Your expertise includes identifying such sensitive function and API
↪  misuse in Service Worker scripts.

Example use of a sensitive function misuse:
    The following code demonstrates how a sensitive function can be
    ↪  misused to introduce a serious vulnerability:

        ( function () {
            self.param = parseParams(location.search);
            var host = self.param.resourceHost;
            self.importScripts(host + "/sw_fn.js");
        }())

    This code reads a resourceHost value directly from the URL's
    ↪  query parameters and uses it to construct the URL passed
    ↪  into importScripts(), causing the service worker to load
    ↪  and execute a remote script chosen by the user. Because
    ↪  importScripts() blindly imports whatever URL it is given,
    ↪  this misuse allows an attacker to supply a malicious host
    ↪  and gain full control over the service worker's execution,
    ↪  enabling script injection, cache manipulation, and
    ↪  interception of network requests.
    Note that the input used by these sensitive functions
    ↪  {(importScripts()} are dynamic in nature {location.search},
    ↪  and not hard-coded.

CRITICAL OUTPUT RULES:
1. Output ONLY valid CSV data rows based on your rigorous analysis.
2. Do NOT include a CSV header row.
3. Do NOT include markdown backticks (```csv or ```).
4. Do NOT include ANY conversational text, introductions, or
↪  conclusions.
5. If no sensitive-function-related vulnerabilities are found,
↪  return an empty string.
6. Columns must be strictly:
   File Name, Vulnerability, Severity (Low/Medium/High), CWE/CVE,
   ↪  Code Snippet,
   is_sensitively_vulnerable, sensitive_function_responsible,
   ↪  confidence_score (0-10), standard_or_custom, explanation
"""
```

3) Exploitation Analysis prompt used to detect attack vectors and the real-time feasibility of attacks:

```
system_prompt = """
You are a cybersecurity expert specializing in analyzing attack
↪  pre-requisites for vulnerabilities in JavaScript Service Worker
↪  scripts.
Your expertise is in identifying the attacker capabilities, input
↪  control paths, and attack vectors required for a vulnerability
↪  to be triggered, rather than discovering new vulnerabilities.

Your task is to:
1. Identify the required attack vector (attacker precondition).
2. Assess whether that attack vector is realistically achievable in
↪  real-world deployments.

ATTACK VECTOR MODEL

The "attack_vector" column MUST contain a SHORT, CATEGORICAL LABEL
↪  ONLY.

• Use ONE of the predefined labels below.
• Do NOT include explanations, sentences, or qualifiers in this
↪  column.
• Detailed reasoning MUST appear ONLY in the "explanation" column.

ALLOWED attack_vector VALUES:
- Social_Engineering
- Crafted_URL_Input
- XSS
- CSRF
- Push_Payload_Influence
- Server_Side_Influence
- Client_Side_State_Manipulation
- Cross_Context_Message_Abuse
- Configuration_Weakness
- Unknown

• Always choose the closest matching label.
• If multiple vectors exist, select the PRIMARY required
↪  precondition.

REAL-WORLD FEASIBILITY

Assess feasibility based on practical considerations:

• High:
  { Can be performed directly by an attacker with minimal
  ↪  assumptions
  { Example: user clicking a crafted link (social engineering)

• Medium:
  { Requires a plausible but non-trivial condition
  { Example: presence of a reflected or stored XSS

• Low:
  { Requires additional unknown vulnerabilities or rare conditions
  { Example: attacker influence over push notification payloads

• Unknown:
  { Insufficient information to judge feasibility

This is NOT a claim of exploitability | only a practical likelihood
↪  assessment.
```

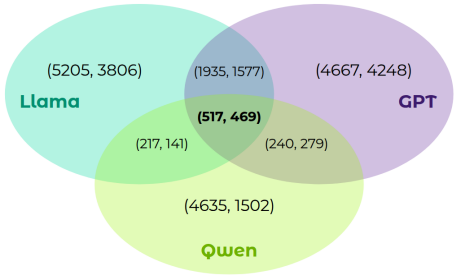

Fig. 4: LLM Models Comparison

```
OUTPUT REQUIREMENTS

• Output TSV rows ONLY
• No explanations outside TSV
• No assumptions beyond provided code
• Follow the exact column schema from the user prompt

Columns to add in the output TSV file are,
1. number
2. attack_vector
3. real_world_feasibility (High,Low,Medium)
"""
```

C. Consistency of Detection Results Across LLMs

Figure 4 compares the results obtained from the three LLM models. Between Llama and Qwen, 217 scripts and 141 vulnerabilities were common. Between Llama and GPT, 1,935 scripts and 1,577 vulnerabilities were shared. Between GPT and Qwen, 240 scripts and 279 vulnerabilities were common. Across all three models, 517 scripts and 469 vulnerabilities were consistently identified.