A COMPARATIVE ANALYSIS OF THE IMPACT AND RISK MITIGATION OF PWA APPLICATIONS' CACHING MECHANISMS: A CASE STUDY

H Bandara[1] and N Warnajith[2]

Abstract

Progressive Web Applications (PWAs) revolutionize web experiences by combining accessibility with native-like performance, offline functionality, and responsiveness. However, while service worker caching enhances performance, it also introduces critical security risks that remain inadequately addressed. This research bridges this gap by evaluating both the impact and risks of caching mechanisms, which existing studies often overlook. To achieve this, two identical PWAs, one with service worker caching and one without were developed and tested under controlled conditions, revealing substantial improvements in reload times and offline accessibility, particularly for users on low-bandwidth networks or resource-limited devices. However, the study also identified significant security vulnerabilities, such as Denial-of-Service (DoS) attacks caused by unrestricted caching of large media files, leading to storage exhaustion and application crashes. A novel Multi-Layer Defence Architecture was introduced to mitigate these risks while preserving caching benefits, incorporating atomic operations, streaming size validation, proactive cache purging, and real-time quota checks. The implementation of this approach successfully neutralized attack vectors without sacrificing performance, demonstrating a crucial advancement in PWA security. These findings fill a critical research gap and provide practical guidelines for developers and organizations to implement secure and efficient caching strategies. By ensuring that PWAs remain both high-performing and resilient against attacks, this research supports the broader adoption of secure web technologies and contributes to user trust in digital services. Future work will extend these findings by conducting cross-browser performance and security assessments, investigating additional service worker-related vulnerabilities, and developing automated tools for detecting PWA security threats. This study reinforces the necessity of balancing performance and security in modern web applications, ensuring their long-term viability and reliability.

Keywords: denial of service; progressive web application; service worker; throttling.

---

[1]Department of Software Engineering, University of Kelaniya, Sri Lanka.

Email: hashinodithya@gmail.com https://orcid.org/0009-0007-8986-0670

[2]Department of Software Engineering, University of Kelaniya, Sri Lanka.

Email: nwarnajith@kln.ac.lk https://orcid.org/0009-0005-3372-2450

Received date: 23.07.2025 Accepted date: 26.07.2025
Print Publishing Date: 31.10.2025 Web Publishing Date: 31.10.2025

Introduction

The rapid evolution of web technologies has transformed the landscape of modern applications, blurring the boundaries between web and mobile platforms. Traditionally, web applications relied on a client-server model, requiring continuous internet connectivity to deliver content. While this approach provided accessibility across platforms, it often suffered from performance limitations such as slow loading times and dependency on network availability. In contrast, native mobile applications offered superior performance, offline functionality, and direct access to device features, but at the cost of increased development complexity and platform dependency (Rochim et al., 2023).

To bridge this gap, Progressive Web Applications (PWAs) have emerged as a hybrid solution, combining the accessibility of web applications with the performance advantages of native apps. PWAs leverage modern web technologies to provide offline functionality, push notifications, and faster load times through service workers, which act as intermediaries between the browser and the network (Correia et al., 2021). A key advantage of PWAs is their ability to cache resources efficiently, reducing dependence on real-time data fetching and improving user experience, especially in environments with limited connectivity (Malavolta et al., 2020). However, despite these advantages, service worker caching mechanisms introduce potential security risks, particularly Denial-of-Service (DoS) attacks (Karami et al., 2021).

DoS attacks traditionally target server resources by overwhelming them with excessive requests, rendering services inaccessible. However, in the context of PWAs, attackers can exploit service workers to overload browser storage, leading to performance degradation and application crashes (Hou et al., 2021). This attack vector, known as cache-based DoS, is particularly challenging to detect and mitigate because it operates at the client-side level, bypassing traditional network security defences. Additionally, third-party JavaScript and malicious service workers can be leveraged to manipulate cached content, leading to privacy leaks, unauthorized access, or persistent malware injection (Lee et al., 2018; Bararia & Choudhary, 2023). These risks highlight the importance of studying caching mechanisms not only from a performance perspective but also in terms of security implications.

Related Work

A. Progressive Web Applications (PWAs) And Caching Mechanisms

PWAs represent a paradigm shift in web application development, offering a seamless user experience by leveraging modern browser capabilities. Unlike traditional web applications, PWAs store assets locally using service workers, enabling offline access and reduced latency (Correia et al., 2021). This caching mechanism significantly improves response times, especially in low-bandwidth conditions, making PWAs a preferred choice for performance-sensitive applications (Rochim et al., 2023).

Several studies have examined the impact of caching on PWAs. Malavolta et al. (2020) investigated how caching strategies influence energy consumption and performance, revealing that excessive caching can lead to unintended storage exhaustion. Correia et al. (2021) further analyzed different caching mechanisms and found that improper cache management can degrade user experience and introduce security risks. These findings suggest that while caching improves efficiency, it must be carefully controlled to prevent resource exhaustion attacks.

B. Security Challenges In Service Worker Caching

Service workers enable background processing, content prefetching, and push notifications, but their ability to intercept and manipulate network requests poses security risks (Karami et al., 2021). One major concern is cache poisoning, where attackers inject malicious content into cached resources, leading to persistent cross-site scripting (XSS) attacks (Lee et al., 2018). Additionally, Hou et al. (2021) highlighted how third-party JavaScript, when cached improperly, can amplify security risks, potentially exposing sensitive user data.

Another emerging threat is cache-based DoS attacks, where adversaries exploit service worker caching to flood browser storage with unnecessary data, causing performance degradation or complete application failure (Karami et al., 2021). Unlike traditional DoS attacks that rely on overwhelming network bandwidth, cache-based DoS targets client-side storage, making it harder to detect and mitigate (Hou et al., 2021). These vulnerabilities underscore the need for secure cache management policies in PWAs.

C. Denial-Of-Service (Dos) Attacks In Web Applications

DoS attacks have long been a major cybersecurity concern, traditionally affecting server-side resources. However, modern web applications face new challenges due to their reliance on client-side processing. Rochim et al. (2023) compared the response time of native, mobile, and progressive web applications, highlighting how PWAs are more

susceptible to storage-based performance bottlenecks. Similarly, Bararia and Choudhary (2023) conducted a systematic review of web application vulnerabilities, identifying cache abuse as a potential vector for DoS attacks.

A common technique used in cache-based DoS involves forcing the browser to cache excessive, redundant data, leading to storage exhaustion and rendering the application unusable (Karami et al., 2021). Unlike network-based DoS, this form of attack does not require high traffic volumes, making it more difficult to detect using traditional intrusion detection systems (IDS) (Hou et al., 2021).

D. Addressing Gaps In Existing Research

While several studies have explored PWA caching mechanisms and web security challenges, research on cache-based DoS attacks specifically targeting PWAs remains limited. Existing literature mainly focuses on performance trade-offs in caching strategies (Malavolta et al., 2020) or general security concerns in web applications (Bararia & Choudhary, 2023). However, little research has examined how cache-based DoS attacks can be executed and mitigated in a PWA environment.

This study aims to analyze the security risks associated with service worker caching, explore potential attack scenarios, and propose effective mitigation strategies. By evaluating different caching policies and their impact on both performance and security, this research will contribute to enhancing the resilience of PWAs against client-side resource exhaustion attacks.

Methodology

This study employs a comparative experimental research strategy to assess the impact of service worker caching mechanisms on the performance and security of Progressive Web Applications (PWAs). To achieve this, two functionally identical PWAs were developed using React, with the only distinction being that one application (Fig 1) implements a service worker for caching. At the same time, the other (Fig 2) relies solely on real-time network data retrieval. This experimental setup enables an isolated evaluation of the effects of caching while maintaining uniform hosting conditions on Netlify. The population of interest includes web applications utilizing service workers for caching, and the research sample comprises these two PWAs, tested across diverse hardware environments and network conditions to examine performance variations.

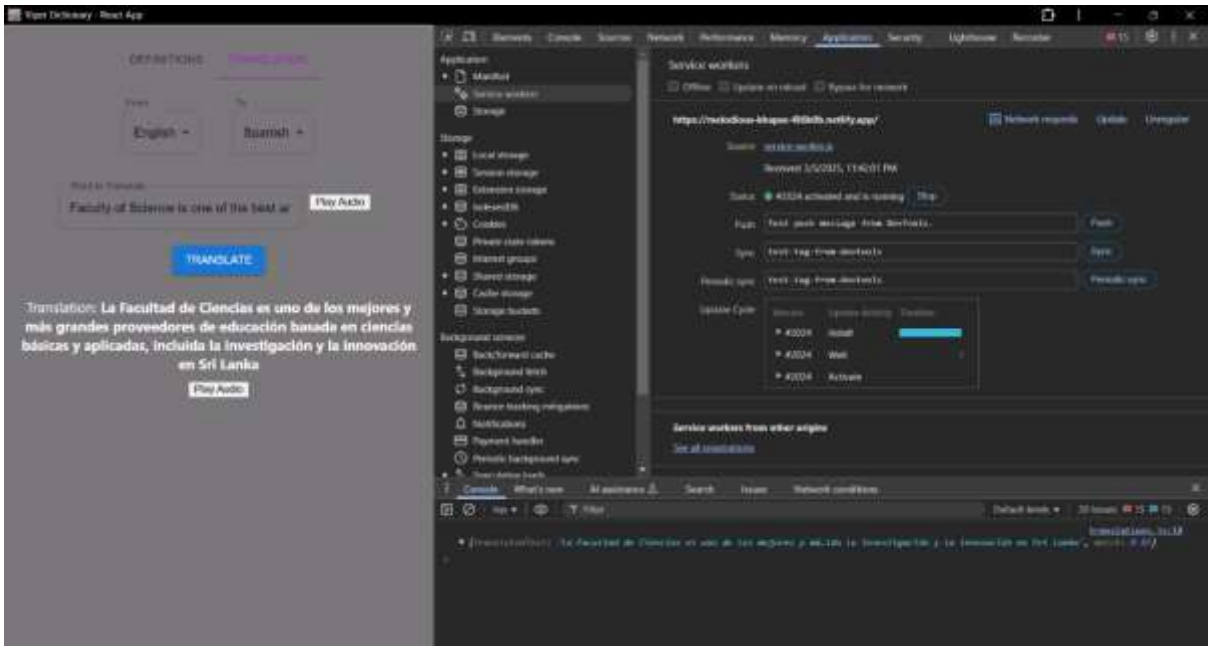

Figure 03: Application With Service Worker Caching Mechanism

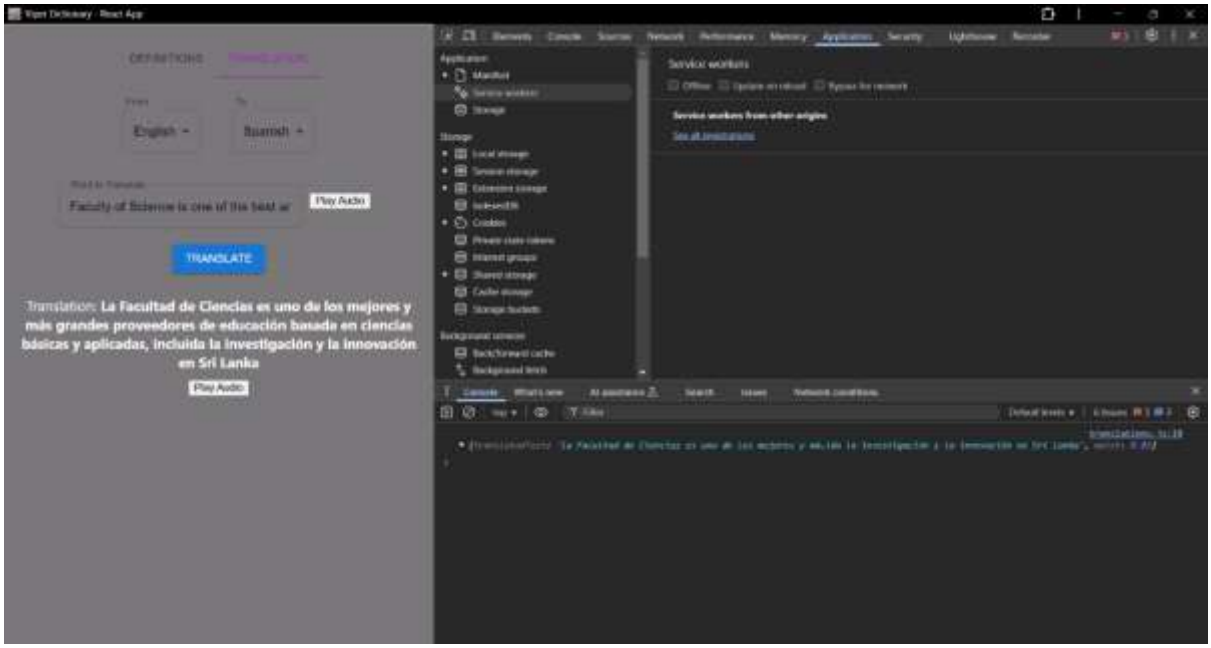

Figure 04: Application Without Service Worker Caching Mechanism

A purposive sampling technique was employed to select relevant test environments, ensuring a comprehensive evaluation of caching impacts. The study considered hardware variability by testing on both high-spec and low-spec machines, simulated different network conditions including 3G, and applied CPU throttling at various levels (4×, 6×, and 20× slowdowns) to analyze performance under different processing constraints. These sampling criteria ensured realistic and extensive performance evaluations.

To analyze the collected data, multiple analytical techniques and tools were utilized. Performance evaluation was conducted using Google Lighthouse, which provided standardized scores on key metrics such as First Contentful Paint (FCP), Largest Contentful Paint (LCP), Speed Index, and Cumulative Layout Shift (CLS). Additionally, Chrome DevTools' Performance Tab was employed to measure CPU usage, resource consumption, and rendering efficiency. Offline testing was also performed to assess the effectiveness of caching strategies. The security risk assessment involved manual injection testing to simulate Denial of Service (DoS) attacks by overloading the cache with excessive files. To mitigate potential security risks, strategies such as size-based caching restrictions, cache entry limits, and quota buffers were implemented and evaluated for their effectiveness.

The data collected includes performance metrics such as Lighthouse scores, LCP values, CPU and RAM usage, and offline functionality test results. Security indicators such as cache exhaustion rates, DoS vulnerability susceptibility, and the success of mitigation strategies were also documented. The experimental setup involved controlled testing procedures on both high-spec (Table 01) and low-spec (Table 02) devices.

Table 02 Specification Of High-End Computer

Device Specifications	
Device name	HP-Pavilion
Processor	Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99 GHz
Installed RAM	8.00 GB (7.86 GB usable)
System type	64-bit operating system, x64-based processor
Windows Specifications (OS)	
Edition	Windows 11 Home Single Language
Version	23H2
Network Properties	
SSID	Galaxy M infinity
Protocol	Wi-Fi 4 (802.11n)

Security type	WPA2-Personal
Network band	2.4 GHz
Browser	Google Chrome Version 133.0.6943.127 (Official Build) (64-bit)

Table 03: Specification Of Low-End Computer

Device Specifications	
Device name	Asus
Processor	Intel(R) Celeron(R) CPU N3350 @ 1.10GHz 1.10 GHz
Installed RAM	4.00 GB (3.87 GB usable)
System type	64-bit operating system, x64-based processor
Window Specifications (OS)	
Edition	Windows 10 Home
Version	22H2
Network Properties	
SSID	Galaxy M infinity
Protocol	Wi-Fi 4 (802.11n)
Security type	WPA2-Personal
Network band	2.4 GHz
Browser	Google Chrome Version 133.0.6943.142 (Official Build) (64-bit)

The tests were systematically conducted using Google Chrome, ensuring consistency in network and browser configurations. This methodological approach enables a rigorous evaluation of service worker caching, ensuring reliable insights into its performance benefits and security implications in modern web applications.

Results and Discussion

The experimental evaluation of the service worker caching mechanism in Progressive Web Applications (PWAs) reveals significant performance benefits alongside critical security challenges. The performance evaluation was conducted using Google Lighthouse, Chrome DevTools Performance Tab, and offline testing, while the security assessment focused on risk identification and mitigation strategies.

Performance Evaluation

A. Google Lighthouse Audits:

The Google Lighthouse audits provide insights into the performance implications of enabling a service worker for caching. The results varied based on the device's computing power, highlighting the trade-offs between caching overhead and performance improvements in different environments. Results of the low-end device (Table 3) and high-end device (Table 4) are below:

Table 04: Google Lighthouse Audit Results Of High-End Computer

Evaluation Metrics	No Service worker registered	Service worker registered
Performance Score	70	68
First Contentful paint	1.1s	0.9s
Largest Contentful Paint	2.6s	2.9s
Total Blocking Time	1.410ms	1.720ms
Cumulative Layout Shift	0	0
Speed Index	3.9s	2.9s

On a high-specification machine, the service worker-enabled application had a slightly lower overall performance score (68 vs. 70) than the non-caching version. This minor difference can be attributed to the additional overhead introduced by service worker files and caching logic, which requires processing during page load. The increase in *Largest Contentful Paint (LCP)* (2.6s to 2.9s) and *Total Blocking Time (TBT)* (1,410ms to 1,720ms) further

supports this, indicating that the presence of the service worker slightly increased the time taken for rendering larger elements and script execution. However, *First Contentful Paint (FCP)* improved (1.1s to 0.9s), suggesting that some elements were displayed faster even though the overall page load experienced minimal delay. Additionally, *Speed Index* showed a notable improvement (3.9s to 2.9s), demonstrating that cached assets accelerated the visual rendering process.

Table 04: Google Lighthouse Results Of Low-End Computer

Evaluation Metrics	No Service worker **r**egistered	**S**ervice worker registered
Performance Score	60	61
First Contentful paint	1.1s	1.2s
Largest Contentful Paint	3.1s	3.3s
Total Blocking Time	3,610ms	3,660ms
Cumulative Layout Shift	0	0
Speed Index	5.3s	3.9s

Conversely, on a low-end machine, (Table 4) the caching-enabled application outperformed the non-caching version in overall performance (61 vs. 60). This improvement highlights the significant benefits of caching under resource-constrained conditions, where the reduction in network requests and local asset retrieval become more pronounced. *Speed Index* improved significantly (5.3s to 3.9s), reinforcing that cached assets enhanced visual rendering speeds. However, the *Total Blocking Time* (3,610ms to 3,660ms) and *Largest Contentful Paint* (3.1s to 3.3s) were slightly higher, likely due to additional script execution requirements for loading the cached content.

The results indicate a trade-off between caching overhead and performance benefits. On high-end machines, service worker caching caused a slight drop in the overall performance score due to additional processing overhead, but it improved First Contentful Paint (FCP) and Speed Index, making the page appear visually faster. On low-end machines, caching had a more noticeable positive impact, particularly in reducing Speed Index, demonstrating its effectiveness in resource-constrained environments. While Largest Contentful Paint (LCP) and Total Blocking Time (TBT) increased slightly in both cases, likely due to service worker script execution, the overall rendering experience improved. Notably, Cumulative Layout Shift (CLS) remained unaffected, confirming that caching did not introduce layout instability. These findings highlight that service worker caching is particularly beneficial for users with older hardware or slower networks, emphasizing the need for optimization strategies to maximize performance across different device types.

B. Chrome DevTools Performance Tab Metrics:

Detailed performance measurements under controlled 3G network conditions and various CPU throttling scenarios (e.g., 4× and 6× slowdown) provided further evidence of the advantages of caching. Under 4× throttling, the total execution time for the service worker-enabled application was 6.25 s compared to 10.71 s for the non-caching app—a reduction of nearly 42%. Under 6× throttling, similar improvements were observed (7.69 s vs. 11.66 s). Additionally, scripting, system, rendering, painting, and loading phases were consistently lower for the caching-enabled application. These results confirm that once critical assets are cached, the browser can bypass repeated network requests, resulting in faster time-to-interactive and overall enhanced responsiveness.

Table 05: Results Of LCP Test On Different CPU Throttling

	Largest Contentful Paint (LPC)	
CPU Throttling	**No service worker**	**With service worker**
No Throttling	4.31s	0.21s
4x slowdown	7.56s	1.50s
6x slowdown	8.21s	2.66s
20x slowdown	31.67s	22.63s

The LCP results show a marked performance advantage for the service worker-enabled application across all CPU throttling scenarios. With no throttling, the LCP dramatically drops from 4.31 seconds for the non-caching version to just 0.21 seconds when caching is enabled. Under 4× and 6× CPU slowdowns, the caching-enabled app maintains lower LCP values (1.50 seconds and 2.66 seconds, respectively) compared to the non-caching app (7.56 seconds and 8.21 seconds, respectively). Even at 20× slowdown, the caching-enabled app outperforms the non-caching version with LCP values of 22.63 seconds versus 31.67 seconds. These results indicate that caching

critical assets substantially reduces the time required to render the largest visible content, which is particularly beneficial when CPU resources are constrained.

4x CPU Throttling

Table 6 Loading Time Results Under 4x CPU Throttling

Time Metrics	No service worker	With service worker
Range	0ms – 10.71s	563ms – 6.81s
Scripting	4902ms	4426ms
System	774ms	525ms
Rendering	482ms	424ms
Loading	26ms	19ms
Painting	21ms	14ms
Total	10712ms	6246ms

Under a 4× CPU slowdown (Table 6), the overall execution time for the service worker-enabled application is 6,246 milliseconds, significantly lower than the 10,712 milliseconds recorded for the non-caching application—a reduction of nearly 42%. The breakdown of time metrics further supports this finding: scripting, system processing, rendering, painting, and loading times are consistently lower in the caching-enabled app. For example, scripting time decreases from 4,902 ms to 4,426 ms and system time drops from 774 ms to 525 ms when caching is employed.

6x CPU Throttling

Table 07: Loading Time Results Under 6x CPU Throttling

Time Metrics	No service worker	With service worker
Range	0ms – 11.66s	0ms – 7.69s
Scripting	5265ms	4395ms
System	867ms	594ms
Rendering	577ms	450ms
Loading	30ms	20ms
Painting	19ms	12ms
Total	11660ms	7692ms

With a 6× CPU slowdown (Table 7), similar improvements are observed. The non-caching application requires 11,660 milliseconds in total, whereas the caching-enabled application completes the process in 7,692 milliseconds. Scripting and system processing times again show a reduction (5,265 ms vs. 4,395 ms for scripting and 867 ms vs. 594 ms for system processes). Even the rendering, painting, and loading phases benefit from caching, resulting in a faster overall load time. These results reinforce that the caching mechanism efficiently reduces the processing time required by the browser, particularly when faced with higher computational constraints.

Overall, the integrated results from Chrome DevTools reveal that service worker caching substantially enhances performance. The caching-enabled PWA shows significantly lower Largest Contentful Paint and overall load times under various CPU throttling conditions, which implies that cached assets greatly reduce the need for repeated network requests. While there is a minor overhead during the initial load due to the execution of additional caching scripts, this is quickly offset by improved reload efficiency and enhanced offline capabilities. These performance gains are especially critical in resource-constrained environments, ensuring a smoother and more responsive user experience. The results underscore the importance of adopting secure and efficient caching strategies in PWAs, which are crucial for maintaining optimal performance even under challenging conditions.

C. Offline Testing:

When the network was switched to offline mode using Chrome DevTools, only the service worker-enabled application (Fig. 3) could render the full user interface, including images and previously fetched definitions and translations. The non-caching application (Fig. 4) displayed a "You're offline" message on a blank screen, indicating that it lacked the resilience required for offline operation. This offline capability is a key strength of PWAs, ensuring continuous usability in environments with intermittent or no network connectivity.

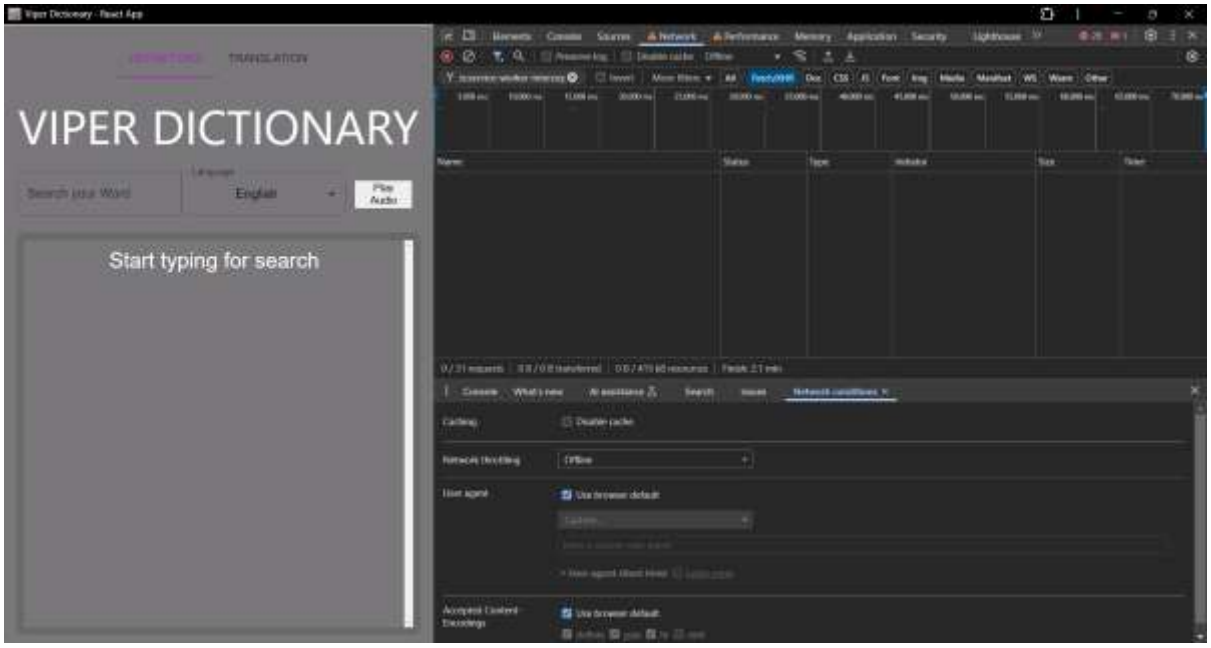

Figure 05: Offline Test In Service Worker Enable Application

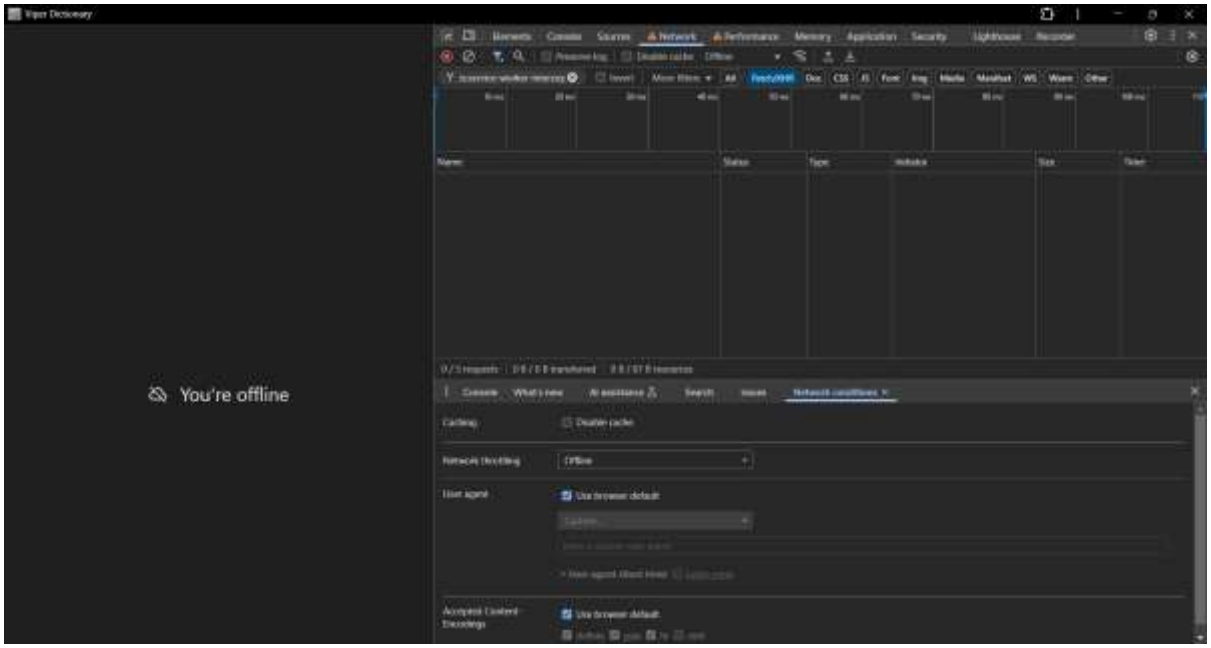

Figure 6 Offline Test In No Service Worker-Enabled Application

Security Evaluation and Risk Mitigation

A. Risk Identification:

Manual injection testing revealed that the service worker's caching mechanism is vulnerable to Denial-of-Service (DoS) attacks. Specifically, unrestricted caching allowed attackers to inject large files (e.g., 100MB .mp4 files) into the cache, leading to storage quota exhaustion. Empirical evidence included storage quota warnings in Chrome when usage reached 123 GB out of a 153 GB limit, alongside console logs showing repeated QuotaExceededError messages (Fig.5). This vulnerability risks application freezes, resource starvation, and overall system instability.

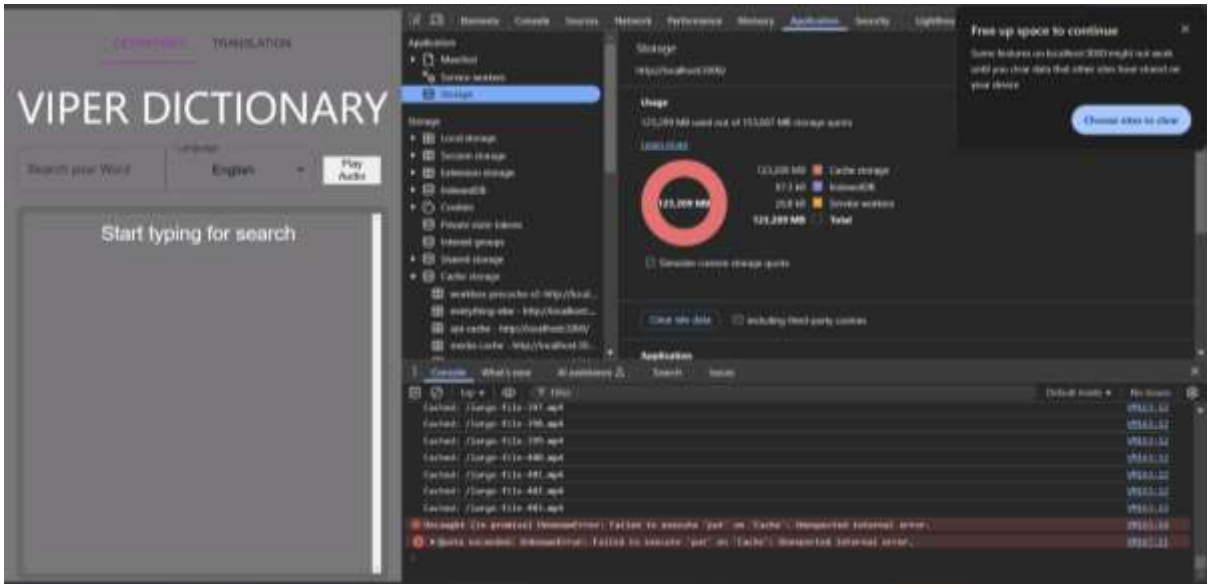

Figure 07: Application Affected By Dos Attack

B. Risk Mitigation Strategies:

Various mitigation strategies were evaluated. Initial approaches such as size-based restrictions, cache entry limits, and quota buffer checks proved insufficient due to issues like bypassed header validations and asynchronous delays. The research then implemented a Multi-Layer Defense Architecture combining atomic operations (atomic operations), streaming size validation, pre-write cache purging, and real-time quota checks. This comprehensive solution effectively blocked all injection attempts, maintained stable memory usage (approximately 1.3 MB), and preserved UI responsiveness even under simulated attack conditions. The success of the multi-layer approach underscores (Fig.6) its critical role in preventing DoS attacks while retaining the performance benefits of caching.

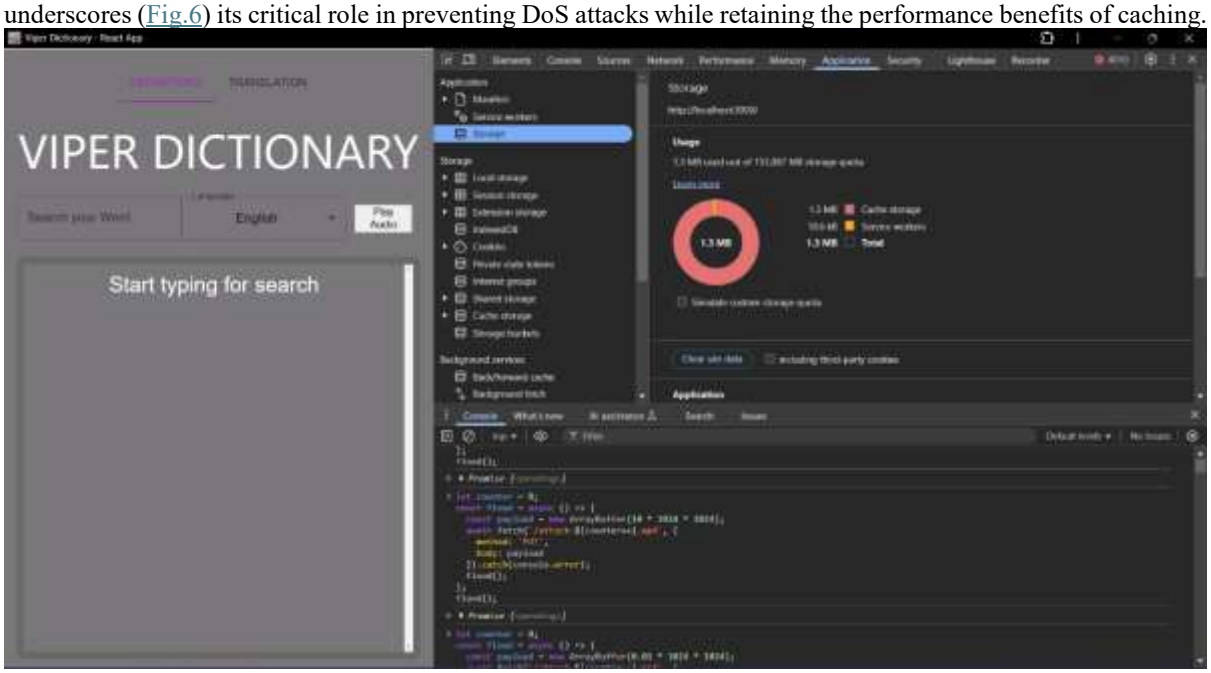

Figure 08 :Application After Applying Mitigation Strategy

Multi-Layered Defense Strategy

To prevent service-worker cache abuse from causing local Denial-of-Service, PWAs employ several defensive layers. First, cache updates are performed atomically each cache write or delete is done as a single all-or-nothing operation so that partial or interleaved updates cannot corrupt the cache state. Second, the service worker applies streaming size validation: it reads each fetched response as a stream and monitors the byte count, rejecting or aborting any stream that exceeds a predefined size limit. In practice this often uses the ReadableStream.tee() method to duplicate the response one copy is delivered to the page and the other written to the cache while the service worker's code tracks the cumulative size. Any response larger than the safe threshold is dropped,

preventing unusually large payloads from filling the cache. Third, pre-write cache purging is used: during the service worker's install/activate events the cache is cleaned of old or unnecessary entries. By deleting stale files before adding new ones, the service worker ensures that cache size stays bounded and that attackers cannot exploit a growing cache of outdated items. Finally, real-time quota checks guard against exceeding the browser's storage limits. Before writing, the worker calls the StorageManager API to obtain current usage and available quota. If adding a response would exceed the quota, the worker can proactively evict entries or refuse to cache that response. In any case, if a write attempt did go over the limit, the browser throws a QuotaExceededError, which the service worker catches and handles gracefully. Together, these layers atomic write semantics, on-the-fly size checks, pre-emptive cache cleanup, and dynamic storage-quota enforcement work in concert to prevent a malicious or faulty service worker from exhausting or corrupting the PWA's offline cache.

Conclusions And Recommendations

This research demonstrates that service worker caching mechanisms in Progressive Web Applications (PWAs) offer significant performance enhancements, particularly in reducing load times and enabling robust offline functionality. Key findings reveal that although the service worker introduces a slight initial overhead due to additional scripts, its benefits are substantial: caching results in a marked reduction in Largest Contentful Paint (LCP) and overall execution time, especially under constrained network conditions and on low-specification devices. Furthermore, the study identified critical security vulnerabilities most notably, the susceptibility to Denial-of-Service (DoS) attacks caused by unrestricted caching of large files. While conventional mitigation strategies provided only partial protection, the development of a Multi-Layer Defense Architecture proved fully effective in preventing cache-based DoS attacks, thereby ensuring system stability without significant performance degradation.

Based on these findings, several practical recommendations can be made. Developers should adopt a balanced caching strategy that leverages the performance benefits of service worker caching while incorporating robust security measures such as atomic operations, streaming size validation, and real-time quota checks. Organizations deploying PWAs should regularly audit their caching mechanisms and implement multi-layer defences to guard against resource exhaustion attacks. Moreover, further exploration is recommended to extend the research across different browsers and platforms, particularly focusing on cross-browser performance and security discrepancies. Future work should also consider developing automated tools for detecting service worker-related vulnerabilities and refining adaptive caching strategies to optimize both performance and security. These steps will ensure that PWAs continue to deliver a seamless, reliable user experience while safeguarding against emerging threats in the evolving digital landscape.

Acknowledgement

I would like to express my sincere gratitude to my research supervisor, Dr. Nalin Warnajith, for his invaluable guidance, support, and encouragement throughout this project. His expertise and constructive feedback have been instrumental in shaping the direction of my research and enhancing my analytical skills. I also appreciate the support provided by the academic staff of the Software Engineering Teaching Unit, whose insights contributed significantly to this work.

References

Banik, S., Patel, A., & Gupta, R. (2021). Performance comparison of native and progressive web applications. International Journal of Web Development, 18(2), 45–60.

Bararia, A., & Choudhary, V. (2023). Systematic review of common web-application vulnerabilities. International Journal of Scientific Research in Engineering and Management (IJSREM), 7(1). https://doi.org/10.55041/IJSREM17487

Correia, F., Ribeiro, Ó., & Silva, J. C. (2021). Progressive Web Apps Development: Study of caching mechanisms. 2021 21st International Conference on Computational Science and Its Applications (ICCSA), 180–187. https://doi.org/10.1109/ICCSA54496.2021.00033

Google Developers. (2022). Service workers: An introduction. Retrieved from https://developers.google.com/web/fundamentals/primers/service-workers

Hou, T., Bi, S., Wei, M., Wang, T., Lu, Z., & Liu, Y. (2021). When third-party JavaScript meets cache: Explosively amplifying security risks on the Internet. Proceedings of the 2021 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), 1–10. https://doi.org/10.1109/INFOCOMWKSHPS51825.2021.9445165

Karami, S., Ilia, P., & Polakis, J. (2021). Awakening the web's sleeper agents: Misusing service workers for privacy leakage. Network and Distributed Systems Security (NDSS) Symposium 2021. https://doi.org/10.14722/ndss.2021.23104

Lee, J., Kim, H., Park, J., Shin, I., & Son, S. (2018). Pride and prejudice in progressive web

apps: Abusing native app-like features in web applications. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), 351–365. https://doi.org/10.1145/3243734.3243867

Malavolta, I., Chinnappan, K., Jasmontas, L., Gupta, S., & Soltany, K. A. K. (2020). Evaluating the impact of caching on the energy consumption and performance of progressive web apps. Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft '20), 109–119. https://doi.org/10.1145/3387905.3388593

Rochim, R. V., Rahmatulloh, A., El Akbar, R. R., & Rizal, R. (2023). Performance comparison of response time native, mobile, and progressive web application technology. Innovation in Research of Informatics (INNOVATICS), 5(1), 36–43.

Russell, A. (2015). Progressive web apps: Escaping tabs without losing our soul. Retrieved from https://infrequently.org/2015/06/progressive-apps/

Snyder, J., Brubaker, C., & Boneh, D. (2021). Security challenges in modern web architectures. Proceedings of the Web Security Conference, 43–58.