RESEARCH ARTICLE

RSP Science Hub

International Research Journal on Advanced Science Hub
2582-4376
www.rspsciencehub.com
Vol. 07, Issue 11 November

http://dx.doi.org/10.47392/IRJASH.2025.110

Client-Side Threats in SPAs: Modeling Security Risks in Popular JavaScript Framework

Miriyala Kiran Kumar[1], *Dr. Kopparthi Suresh*[2]
[1]*PG Scholar, Bhimavaram Institute of Engineering and Technology, Pennada*
[2]*Professor and Principal, Bhimavaram Institute of Engineering and Technology, Pennada*
Emails: *kkm2404@gmail.com*[1], *sureshkgrl@gmail.com*[2]

Article history
Received: 17 September 2025
Accepted: 09 October 2025
Published: 25 November 2025

Keywords:
Client-side Security, React, Vue.js, Single Page Applications (SPAs Threat Modeling, Cross-site scripting, (XSS), Web security assessment, JWT.

Abstract
Single Page Applications (SPAs) have reshaped web development by improving responsiveness and interactivity, but the shift of application logic and data handling to the client side has introduced security challenges that traditional server-centric models do not adequately address. This study proposes and validates a threat model specifically designed for SPAs, focusing on two widely adopted JavaScript frameworks, React and Vue.js. Two prototype applications with equivalent functionality were developed and evaluated using a modified STRIDE methodology, combining static analysis tools (ESLint, SonarQube, Snyk), dynamic testing tools (OWASP ZAP, Burp Suite), and manual inspection of client-side code and runtime behavior. The analysis identified common vulnerabilities across both frameworks, including DOM-based XSS, insecure token storage, broken route guards, and exposed API endpoints. React showed higher risk when unsafe rendering practices such as dangerously SetInnerHTML were used, while Vue's vulnerabilities were linked to insecure use of v-html and un validated dynamic imports. Mitigation strategies, including input sanitization, Http Only cookie-based token storage, Content Security Policy (CSP), and strict route guards, significantly reduced vulnerabilities. This work delivers a structured SPA-specific threat model and reproducible methodology, providing developers and security practitioners with actionable guidance for building more secure client-side applications.

1. Introduction

Single Page Applications (SPAs) have transformed modern web development by delivering faster, more seamless user experiences compared to traditional Multi-Page Applications (MPAs). Instead of reloading entire pages, SPAs load a single HTML document and dynamically update content using JavaScript and AJAX. Frameworks like React and Vue.js have become the preferred tools for building these high-performance, interactive web apps. However, this shift to client-side rendering introduces new security challenges [21]. SPAs rely heavily on browser-based logic, client-side routing, and APIs for crucial tasks such as authentication and data management. While these features improve usability and efficiency, they also increase the application's attack surface. Common vulnerabilities in SPAs include DOM-based Cross-Site Scripting (XSS), token leakage, exposure of API endpoints, and broken access control due to inadequate route protection. Traditional web security [22] models and testing tools often fall short when applied to SPAs. Methods like STRIDE may not fully address the dynamic behavior and unique architecture of modern JavaScript frameworks. Similarly, security scanners like OWASP ZAP struggle to effectively analyze SPA content and client-side logic. This research addresses these challenges by creating a client-side threat model tailored specifically for SPAs, with a comparative focus on React and Vue.js. It analyzes how these frameworks' design decisions, templating, and data binding mechanisms influence security risks [23]. Using practical testing and code analysis, this study uncovers how SPAs handle or fail to handle key vulnerabilities, professionals with a clear threat model, a comparative framework analysis, and actionable mitigation strategies. By doing so, it bridges the gap between traditional threat modeling and the complex security needs of modern SPA architectures. Development and validation of a client-side threat model tailored to SPA architecture. Framework-specific vulnerability assessment of React and Vue.js.Real-world security testing using static analysis tools (ESLint, SonarQube) and dynamic testing tools (OWASP ZAP, Burp Suite). Implementation of effective mitigation strategies for common client-side security risks. Proposal of a reusable, framework-agnostic methodology for frontend security evaluation in JavaScript-based SPAs.

2. Literature Review

Single Page Applications (SPAs) represent a significant evolution in web application architecture. With client-side frameworks such as React and Vue.js, modern applications perform much of their logic on the frontend. This architectural shift has introduced new client-side security challenges that traditional threat models and server-focused mitigation strategies fail to fully address. This section reviews existing literature on web application security, SPA-specific vulnerabilities, framework-specific risks, and threat modeling approaches relevant to this study. Web security has traditionally concentrated on server-side vulnerabilities such as SQL Injection, Cross-Site Scripting (XSS), and authentication flaws, as outlined in the OWASP Top 10 [14]. However, the architectural transition to Single Page Applications (SPAs) has shifted much of the attack surface to the client side, exposing new risks. Early studies highlighted how XSS evolved into DOM-based variants driven by JavaScript document modifications, making detection harder with traditional methods [12], [13]. The growing complexity of frontend-heavy systems was further emphasized by Kornienko et al. [12] and Christakis et al. [13], who showed that static-only or server-focused threat models fail to capture SPA-specific vulnerabilities. Framework-level behaviors significantly affect security posture. React's JSX provides safe defaults but becomes dangerous when developers use insecure functions such as dangerouslySetInnerHTML [4]. Similarly, Vue.js escapes content by default, yet its v-html directive and unsafe plugin integration enable content injection [8]. Comparative studies confirmed that both frameworks are highly dependent on developer practices rather than inherent safeguards [3], [8]. Case analyses by RadixWeb [4] showed recurring React risks in state handling, weak authentication flows, and insecure dependencies, while Hellquist [3] identified framework-level weaknesses tied to misused libraries and poor sanitization. Industry reports validate these risks. ENISA [5] and Veracode [10] highlighted insecure client-side routing, token mismanagement, and API misuse as recurring vulnerabilities in modern web applications. MITRE's CWE Top 25 [6] underscored that critical weaknesses like improper input handling and insufficient authentication are directly exploitable in SPAs. Similarly, the Snyk JavaScript Security Report [7] and OWASP AppSec 2023 guidance [9] warned of insecure token storage in localStorage, dependency-based vulnerabilities, and poor Content Security Policy (CSP) enforcement. Threat modeling approaches have evolved accordingly. Microsoft's STRIDE model provided a foundation [14], but researchers such as Gupta et al. [11] proposed hybrid SPA-specific models incorporating Data Flow Diagrams (DFDs), attack trees, and runtime analysis. Practical DevSecOps [1] and ISACA [2] further recommended embedding threat modeling into DevSecOps pipelines, ensuring risks are continuously reassessed during agile development cycles. OWASP [9] added SPA-specific risks like API fuzzing, Broken Object Level Authorization (BOLA), and insecure deserialization. Babaey and Ravindran [20] extended this by proposing AI-driven frameworks for detecting evolving cross-site scripting (XSS) patterns in client-side applications. Recent literature and industry updates between 2024 and 2025 underscore an escalating threat landscape. ENISA [5] highlighted supply chain vulnerabilities within frontend ecosystems, while the Security Industry Association (SIA) [19] identified client-side attack vectors as top security megatrends. Wijckmans [17] and TechRadar/c/side [18] documented quarterly attack reports showing a surge in DOM-based XSS and token theft in production SPAs. Cohen [16] provided a novel framework for browser security posture analysis, demonstrating the importance of runtime protections like CSP and Subresource Integrity (SRI) [15]. Testing methodologies align with this hybrid approach. OWASP ZAP and Burp Suite remain essential runtime analysis tools but often require headless browsing to properly evaluate JavaScript-heavy interfaces [10]. Static tools like ESLint, SonarQube, and Snyk flag unsafe coding patterns [7], while runtime defenses such as CSP and SRI [15] offer added hardening layers. Christakis et al. [13] argued that only composite approaches—merging static, dynamic, and runtime analysis—can holistically address SPA vulnerabilities. Identified Gaps in Literature: Despite growing recognition of SPA-specific risks, few studies directly compare React and Vue.js under controlled experimentation. Existing reports [1], [2], [5]–[7], [19] stress the need for reusable threat models tailored to SPAs that integrate with DevSecOps workflows. Specific gaps include the lack of systematic mappings between SPA behaviors (e.g., token storage, route guards, and dynamic imports) and known attack vectors. This study builds upon these gaps by providing a comparative, framework-specific threat model and empirical evaluation of React and Vue.js.

3. Methodology

This chapter outlines the research methodology used to analyze and model client-side security threats in Single Page Applications (SPAs) developed with two widely adopted JavaScript frameworks: React and Vue.js. The approach involves creating controlled test environments, implementing comparable sample applications, applying tailored threat modeling techniques, selecting appropriate security assessment tools, and establishing evaluation metrics for vulnerability detection and mitigation. The process, summarized in Fig. 1, follows a structured workflow that begins with experimental setup and progresses through application development, threat modeling, security testing, mitigation, and comparative evaluation. This methodology provides a rigorous and reproducible framework for the comparative security [24] analysis of SPA architectures. Research Design: The study adopts a comparative experimental design that integrates qualitative threat modeling with quantitative vulnerability assessment. Two SPAs—one built with React and the other with Vue.js—were developed to deliver equivalent functionality, ensuring an objective and consistent basis for security comparison. Core features implemented in both applications include client-side routing, JWT-based authentication, RESTful API integration, and user input handling. The methodology proceeds through five sequential phases (illustrated in Figure. 1): Application Development (React and Vue), Threat Modeling, Security Testing and Analysis, Mitigation Strategy Formulation, Comparative Evaluation Each phase is designed to systematically build, analyze, and evaluate the security [25] posture of the two frameworks.

Development of Sample Applications: Two prototype SPAs were created:

Application 1 (React-based): Built with React 18, using React Router for navigation, Context API for state management, and fetch API for server communication.

Application 2 (Vue-based): Developed using Vue 3, Vue Router, and Vuex for state management, replicating the frontend logic and API interactions of the React application. Both applications implement, JWT-based authentication flows, protected client-side routes, User input forms with validation, API-driven dashboard rendering, Integration with a simulated Node.js/Express REST API backend Functional and architectural parity was maintained to isolate framework-specific security factors.

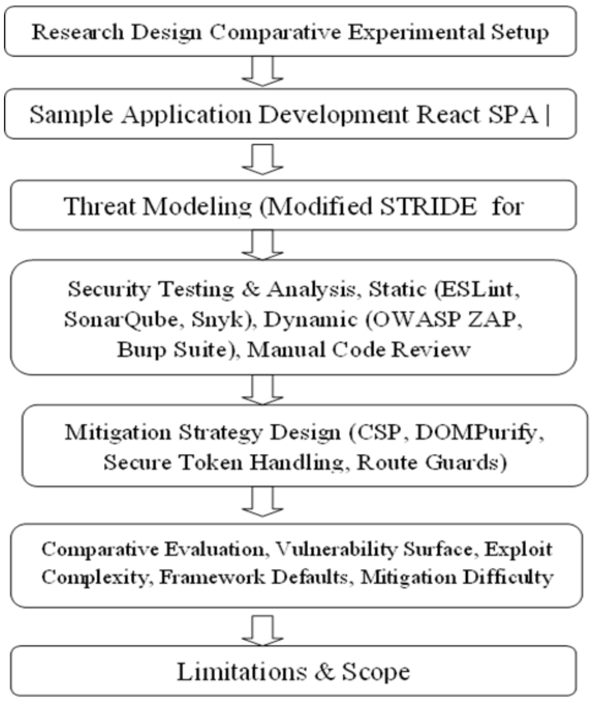
Research Design Comparative Experimental Setup
Sample Application Development React SPA |
Threat Modeling (Modified STRIDE for
Security Testing & Analysis, Static (ESLint, SonarQube, Snyk), Dynamic (OWASP ZAP, Burp Suite), Manual Code Review
Mitigation Strategy Design (CSP, DOMPurify, Secure Token Handling, Route Guards)
Comparative Evaluation, Vulnerability Surface, Exploit Complexity, Framework Defaults, Mitigation Difficulty
Limitations & Scope

Figure 1 Workflow of the Research Methodology

3.1. Threat Modeling Approach

A modified STRIDE framework, adapted for SPA-specific client-side threats, was used to systematically identify vulnerabilities: Spoofing: Token forgery and identity manipulation. Tampering: Modification of API responses via browser developer tools. Repudiation: Lack of effective frontend-side logging or audit trails. Information Disclosure: Data leakage through browser tools or insecure storage. Denial of Service: Client-side resource exhaustion impacting availability. Elevation of Privilege: Unauthorized route or feature access due to insufficient client-side controls. For each threat, analysis considered the default framework security posture, typical developer implementation patterns, and the feasibility of real-world exploitation. Threat modeling outputs include threat catalogs and detailed client-server interaction maps.

3.2. Security Testing Tools and Techniques

Static Analysis: ESLint (with security-specific plugins) for detecting unsafe coding patterns. SonarQube for identifying insecure API usage and code vulnerabilities Snyk CLI for scanning dependency vulnerabilities [26].

Dynamic Analysis: OWASP ZAP for runtime security testing against XSS, insecure APIs, and authentication flaws. Burp Suite for manual analysis of route manipulation, API probing, and token handling.

Manual Code Review: Inspection of potentially unsafe functions like React's dangerouslySetInnerHTML and Vue's v-html. Examination of token storage approaches (e.g., localStorage, sessionStorage). Verification of client-side route guard implementations and lazy loading mechanisms.

Mitigation Strategy Design: Identified vulnerabilities were addressed by applying mitigation strategies derived from OWASP guidelines, official framework best practices, and security standards: Security headers configuration (Content Security Policy, X-Frame-Options). HTML sanitization using DOMPurify. Secure token storage via HttpOnly cookies. Robust client-side route guard implementation and optimized lazy loadingAfter mitigation implementation, both SPAs underwent retesting to validate the effectiveness of the security measures. Iterative adaptations were made based on retest results.

Comparative Evaluation Criteria: Security postures of React and Vue applications were compared along four dimensions: Vulnerability Surface Area: Number of identified threats. Exploit Complexity: Effort required to successfully carry out an attack. Framework Default Security: Built-in prevention and security mechanisms. Clear depiction of the flow from experimental setup to evaluation and limitations [27]. Distinct color coding that aids in understanding the stages.Inclusion of both static and dynamic analysis, manual review, and mitigation strategies. Comprehensive evaluation criteria covering security and practical aspects

4. Results and Analysis

This section summarizes the results of threat modeling and security testing of the React- and Vue-based SPAs. Findings are categorized by vulnerability type, framework behavior, and mitigation feasibility, integrating both tool-based and manual analyses.

4.1. Threat Identification Summary

The STRIDE model revealed common client-side vulnerabilities in both frameworks Shown in Table 1 Threats Identified per Framework.

Table 1 Threats Identified per Framework

Threat Category	React	Vue.js
JWT Tampering	5	4
Route Guard Bypass	6	5
Information Disclosure	4	3
DOM-based XSS	7	6
Broken Access Control	5	5
Token Storage Risk	6	6
Content Injection	4	2
Total Vulnerabilities	**37**	**31**

The STRIDE threat model identified several vulnerabilities across both frameworks. Tab. I summarize the findings, with values indicating the number of confirmed instances observed during testing.

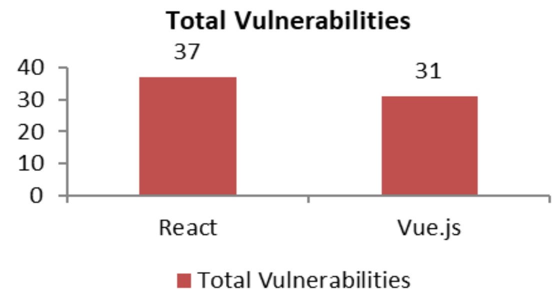

Figure 2 Total Vulnerabilities per Framework

Figure. 2 presents the comparison of total vulnerabilities identified in React and Vue.js. React exhibited approximately 37 vulnerabilities, whereas Vue.js showed about 31 vulnerabilities. These results indicate that React had a broader vulnerability surface than Vue.js in the security evaluation.

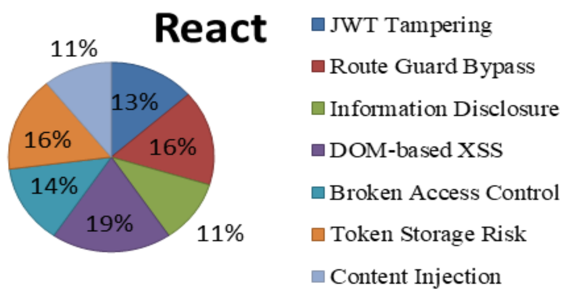

Figure 3 Distribution of Threat Categories(React)

Figure. 3 illustrates the distribution of vulnerabilities identified in the React-based SPA. DOM-based XSS (19%) and route guard bypass (16%) represent the most critical issues, followed by token storage risks (16%) and broken access control (14%). JWT tampering, information disclosure, and content injection each contribute between 11–13%, highlighting that multiple moderate risks collectively broaden the overall attack surface.

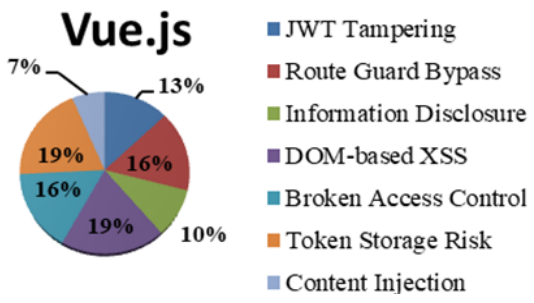

Figure 4 Distribution of Threat Categories(Vue)

Figure. 4 illustrates the distribution of vulnerabilities in Vue.js across different threat categories. The highest proportions are DOM-based XSS and Token Storage Risk, each accounting for 19%, followed closely by Route Guard Bypass and Broken Access Control at 16% each. Information Disclosure (10%) and Content Injection (7%) were less frequent, indicating that Vue.js is more exposed to cross-site scripting and token handling weaknesses than to injection-based threats.

Table 2 Dynamic Analysis Findings

Test Category	React	Vue.js
DOM-based XSS	7	6
API Exposure	5	5
Route Manipulation	6	5
Session Management	5	5
CSP Missing	4	4
Total Findings	**27**	**25**

Table. 2 summarizes the dynamic analysis findings for React and Vue.js. React recorded a slightly higher number of vulnerabilities (27) compared to Vue.js (25), with notable differences in DOM-based XSS (7 vs. 6) and route manipulation (6 vs. 5). Overall, both frameworks showed similar weaknesses in API exposure, session management, and CSP misconfigurations, indicating shared challenges in runtime security.

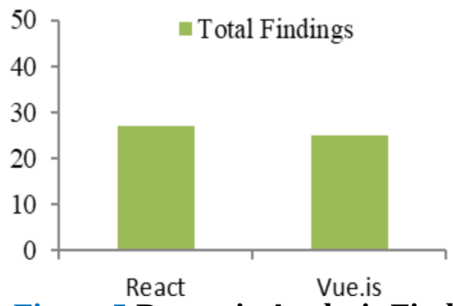

Figure 5 Dynamic Analysis Findings

Figure. 5 illustrates the chart illustrates the total number of vulnerabilities identified through dynamic analysis in React and Vue.js applications. React recorded 27 findings, slightly higher than Vue.js, which showed 25 findings. This outcome indicates that both frameworks exhibit comparable runtime security weaknesses, with React presenting a marginally larger vulnerability surface [28].

Comparative Framework Behavior: React requires more manual sanitization and careful handling of JSX, while Vue provides safer defaults but is weakened by v-html or unsafe imports. Neither framework enforces secure token storage or CSP by default.

Mitigation and Re-Evaluation: Mitigation included DOMPurify, HttpOnly cookies, CSP headers, and stronger route guards. Post-implementation, DOM-based XSS and unauthorized access were eliminated, and token theft risk reduced.

Conclusion and Future Work

The adoption of Single Page Applications (SPAs) has greatly enhanced web performance, interactivity, and user experience. However, the shift to client-side rendering and routing introduces new security challenges that traditional threat models do not fully address. This research conducted a comparative security analysis of React and Vue.js SPAs using a structured methodology combining threat modeling, static and dynamic analysis, and manual inspection. Both frameworks exhibited common vulnerabilities such as DOM-based Cross-Site Scripting (XSS), broken access control, and insecure token storage. React's dangerouslySetInnerHTML was identified as a particularly high-risk feature, while Vue.js, despite safer defaults, remained vulnerable through usage of v-html and unvetted third-party components. Security depended heavily on developer practices rather than framework defaults.Mitigation strategies—enforcing Content Security Policy (CSP), input sanitization with DOMPurify, secure routing, and storing authentication tokens in HttpOnly cookies—effectively reduced attack surfaces. This study highlights the need for SPA-specific security models, improved developer guidance, and stronger framework defaults to tackle client-side risks. The contribution includes a validated SPA threat model and a framework-agnostic methodology to assist developers and security teams in building more secure client-side applications.

Future Work

Future research can extend this study by including additional SPA frameworks such as Angular, Svelte, and Next.js for broader comparison. Developing automated tools for SPA-oriented threat modeling and using AI or machine learning to detect client-side anomalies in real time are promising directions. Further investigation into the security of Progressive Web Apps (PWAs) and mobile SPAs—especially regarding service workers and offline storage—would also enhance understanding. Finally, embedding security scanning and threat modeling into CI/CD pipelines will be essential for maintaining continuous and scalable web application security.

References

[1]. Practical DevSecOps, Threat Modeling Best Practices for 2025. 2025.
[2]. ISACA, Threat Modeling Revisited. ISACA White Paper, 2025.
[3]. E. Hellquist, “Evaluating Security for JavaScript-based Frontend Frameworks,” M.S. thesis, Umeå Univ., Umeå, Sweden, 2024.
[4]. RadixWeb, React JS Security Vulnerabilities: Identify and Fix Common Threats. RadixWeb Report, 2024.
[5]. European Union Agency for Cybersecurity (ENISA), ENISA Threat Landscape 2024. ENISA, 2024.
[6]. MITRE, CWE Top 25 Most Dangerous Software Weaknesses, 2024. MITRE Corporation, 2024.
[7]. Snyk, State of JavaScript Security 2023. Snyk Security Report, 2023.

[8]. Land2Cyber, “Comparative Study of SPA Framework Vulnerabilities,” Cybersecurity Review, 2023.

[9]. OWASP, SPA Security Best Practices. OWASP AppSec Global Conf., 2023.

[10]. Veracode, State of Software Security 2022. Veracode Security Report, 2022.

[11]. V. Gupta, A. Sharma, and R. Kumar, “Hybrid Threat Modeling for SPAs,” Int. J. Comput. Appl., vol. 184, no. 23, pp. 15–22, 2022.

[12]. A. Kornienko, P. Müller, and S. Meier, “Modern SPA Architecture and Security Implications,” J. Web Eng., vol. 20, no. 4, pp. 299–317, 2021.

[13]. M. Christakis, N. Polikarpova, and P. Müller, “Integrating Static and Dynamic Analysis for SPA Security,” IEEE Trans. Softw. Eng., vol. 47, no. 6, pp. 1123–1138, Jun. 2021.

[14]. OWASP, OWASP Top 10: 2021. OWASP Foundation, 2021.

[15]. W3C, Subresource Integrity (SRI) Specification. W3C Recommendation, 2016.

[16]. A. Cohen, “Browser Security Posture Analysis: A Client-Side Security Assessment Framework,” arXiv preprint arXiv:2505.08050, 2025.

[17]. S. Wijckmans, “Client-Side Attack Recap – Q1 2025,” c/side Threat Research Team Blog, 2025.

[18]. TechRadar/c/side, “Client-Side Attack Report Q2 2025 Highlights,” TechRadar Pro Security News, Aug. 2025.

[19]. Security Industry Association (SIA), 2025 Security Megatrends. SIA Report, 2024.

[20]. V. Babaey and A. Ravindran, “GenXSS: An AI-Driven Framework for Automated Detection of XSS Attacks in WAFs,” arXiv preprint arXiv:2504.08176, 2025.

[21]. B. S. Murthy, R. R. PBV, M. Prasad, P. K. Sree, P. J. R. S. Raju and K. S. Kumar, "Hybrid Security Framework and Machine Learning Based Anomaly Detection for Machine-to-Machine communications, " 2025 International Conference on Computational Robotics, Testing and Engineering Evaluation (ICCRTEE), Virudhunagar, India, 2025, pp. 1-5, doi: 10.1109/ICCRTEE64519.2025.11053023.

[22]. Prasad, M. et al. (2024). Robust Strategies for Authenticating and Exchanging Secret Keys in Machine-to-Machine Communications with Enhanced Security. In: Bhateja, V., Lin, H., Simic, M., Attique Khan, M., Garg, H. (eds) Cyber Security and Intelligent Systems. ISDIA 2024. Lecture Notes in Networks and Systems, vol 1056. Springer, Singapore. https://doi.org/10.1007/978-981-97-4892-1_18

[23]. A. Rapaka, M. Prasad, R. R. Pbv, P. S. Murty, and K. S. Pokkuluri, “Enhancing Network Security: Leveraging Machine Learning for Intrusion Detection,” Journal of Electrical Systems, vol. 20, no. 2, pp. 1555-1562, 2024, doi: 10.52783/jes.1460.

[24]. Satyanarayana Murty, P.T., Prasad, M., Raja Rao, P.B.V., Kiran Sree, P., Ramesh Babu, G., Phaneendra Varma, C. (2023). A Hybrid Intelligent Cryptography Algorithm for Distributed Big Data Storage in Cloud Computing Security. In: Morusupalli, R., Dandibhotla, T.S., Atluri, V.V., Windridge, D., Lingras, P., Komati, V.R. (eds) Multi-disciplinary Trends in Artificial Intelligence. MIWAI 2023. Lecture Notes in Computer Science(), vol 14078. Springer, Cham. https://doi.org/10.1007/978-3-031-36402-0_59

[25]. A. Mallikarjuna Reddy, K. Srinivas Reddy, M. Prasad, and A. Obulesh. 2021. Internet of things (IoT) security threats and countermeasures. Netw. Secur. 5 (2021), 12–26.

[26]. Satti, S.K., Suganya Devi, K., Muppalaneni, N.B., Maddula, P. (2025). Real-Time Surveillance System to Monitor Vehicles and Pedestrians for Road Traffic Management. In: Maryam, H., Malik, M.M., Khan, I.U., Gupta, S.K. (eds) AI-Driven Transportation Systems: Real-Time Applications and Related Technologies. Information Systems Engineering and Management, vol 62. Springer, Cham. https://doi.org/10.1007/978-3-031-98349-8_10

[27]. Satti, Satish Kumar, Prasad Maddula, and NV Vishnumurthy Ravipati. "Unified

approach for detecting traffic signs and potholes on Indian roads." Journal of King Saud University-Computer and Information Sciences 34.10 (2022): 9745-9756.

[28]. S. K. Satti, G. N. V. Rajareddy, P. Maddula and N. V. Vishnumurthy Ravipati, "Image Caption Generation using ResNET-50 and LSTM," 2023 IEEE Silchar Subsection Conference (SILCON), Silchar, India, 2023, pp. 1-6, doi: 10.1109/SILCON59133.2023.10404600.