Journal homepage: https://itce.vn.ua/en

Information Technologies and Computer Engineering **Vol. 22, No. 3. 2025**

Article's History: Received: 02.09.2025 Revised: 10.11.2025 Accepted: 23.12.2025

UDC 004.056.5:004.738.5:004.942 DOI: 10.31649/vitce/3.2025.09

Tamper-resistant architecture of Server-Driven UI with real-time Merkle proof verification

Vladyslav Ananchenko*

Postgraduate Student
Academician Stepan Demianchuk International University of Economics and Humanities
33000, 4 Stepan Demyanchuk Str., Rivne, Ukraine
https://orcid.org/0009-0004-8963-775X

Yurii Lotiuk

PhD in Pedagogical Sciences, Associate Professor
Academician Stepan Demianchuk International University of Economics and Humanities
33000, 4 Stepan Demyanchuk Str., Rivne, Ukraine
https://orcid.org/0000-0001-6696-5583

Abstract. Server-driven user interface systems require protection from unauthorised modifications to ensure the integrity and security of displayed data. The purpose of this study was to develop a cryptographically verifiable change log of the user interface for systems with a Server-Driven User Interface. Within the study, methods of theoretical modelling, experimental testing, software implementation, and analysis of the regulatory framework were applied to design, verify, and evaluate a cryptographic change log in a client-interface environment. The main results showed that the use of signed structured interface blocks with hashing and digital signatures ensured the impossibility of undetected interface modification on the client side. Construction of the change log based on a hash tree guaranteed authenticity, immutability, and cryptographic verification of each interface element even under complex distributed conditions. Integration with advanced React rendering mechanisms enabled real-time verification of interface authenticity, ensuring compliance with international standards for personal data protection and transaction security. Furthermore, the results showed that client verification of Merkle proofs for blocks in React detected modifications before rendering, with an average verification time of 0.328 milliseconds per block. Auditing of blueprint file changes and the publish-subscribe system ensured data traceability and relevance, while component rendering after updates lasted only 2.7 milliseconds for the main component and 0.4 milliseconds for the button. Experiments confirmed a 94% attack-blocking rate, a reduction in rendering latency (from 850 to 300 milliseconds under slow network conditions), and a cache hit rate maintained at 94-95% under low load. Combined with improved key interface-interaction metrics, these results demonstrate the effectiveness of the proposed architecture. The obtained findings may be used by developers of critical web applications to implement secure interfaces that verify integrity in real time and comply with international security requirements

Keywords: cryptographically verifiable change log; principle of non-repudiation; evolution of React; overhead evaluation; cache hit rate

Introduction

The Server-Driven User Interface (SDUI) assumes dynamic generation of interface structures based on configuration data received from the server. This approach enables centralised adaptation of the interface to the user context (role, language, access rights, etc.), but simultaneously introduces new risks: the client visualises data whose authenticity it cannot independently verify. Vulnerability to attacks through configuration tampering, supply-chain compromise, or internal server errors threatens the integrity and security of critical interfaces. The research problem lies in the absence of mechanisms that allow the client side to independently verify the authenticity of received interface

Suggested Citation:
Ananchenko, V., & Lotiuk, Yu. (2025). Tamper-resistant architecture of Server-Driven UI with real-time Merkle proof verification. *Information Technologies and Computer Engineering*, 22(3), 9-22. doi: 10.31649/vitce/3.2025.09

*Corresponding author

Copyright © The Author(s). This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/)

structures before rendering, without relying on delivery infrastructure or server environment. This creates a protection gap in dynamic interfaces, which must be closed by integrating verifiable integrity verification mechanisms.

The study by A.J. Undirwadkar (2025) emphasised the advantages of the SDUI architecture, particularly its ability to provide dynamic interface updates without redeployment, accelerating development cycles and ensuring cross-platform consistency. A. Orynchak *et al*. (2024) explored JavaScript capabilities for real-time monitoring and threat response, demonstrating the efficiency of logging in protecting web applications – relevant for client-side interface integrity verification. Moreover, L. Shport (2025) analysed payment-system vulnerabilities and proposed a cryptographic architecture with encryption, authentication, and auditing modules ensuring multi-level protection of dynamic interfaces.

H.B. Azhar *et al*. (2025) proposed a post-quantum structure – Quantum-Resistant Merkle Trees – combining Zero-Knowledge Scalable Transparent Argument of Knowledge, lattice cryptography, and randomised hash functions to enhance security and performance. Experiments demonstrated a 28-32% reduction in proof-generation time compared to classical Merkle trees while maintaining logarithmic verification complexity. The results of O. Patel (2022) showed that Merkle proofs ensured transaction integrity with zero disclosure, reducing computational costs and increasing confidentiality in distributed systems. In turn, P. Du *et al*. (2022) proposed a tamper-resistant data-query model based on B+ and Merkle trees, ensuring the authenticity of query results in blockchain systems and improving reliability and security of dynamic data. Furthermore, S. Ridhorkar & S. Mishra (2024) developed a secure blockchain-based system for digital asset and inheritance management using Quantum-Resistant Dilithium Signatures and Merkle trees to ensure immutability, transparency, and protection from quantum-computer attacks, strengthening trust in automated processes within Decentralised Applications.

In addition, M. Havatiuk & I. Saiapina (2025) proposed an efficient graphical-interface update mechanism based on reactive programming, virtual Document Object Model (DOM), and centralised state management, which reduced computational costs and improved update speed. In the study by O. Kuznetsov *et al*. (2025), a secure generation and verification system for QR codes was developed using digital watermarks and a neural-network authentication model, ensuring high accuracy in forgery detection in dynamic environments. Moreover, M.T. Rubel *et al*. (2025) examined the use of blockchain technologies for automating real-time financial reconciliation, reducing error risks and meeting regulatory requirements such as those of the Securities and Exchange Commission and the Public Company Accounting Oversight Board.

Thus, existing studies do not address mechanisms for client-side interface-integrity verification before rendering in SDUI using signed JSON (JavaScript Object Notation) and Merkle proofs combined with React and change auditing. Therefore, the present study aimed to create a Cryptographically Verifiable Change Log (CVCLog) for user interfaces in SDUI architecture systems. The research objectives included proposing a modification-resistant interface architecture based on signed JSON packets organised into a Merkle tree, as well as implementing client-side integrity-proof verification before rendering in React and API (Application Programming Interface) level interface-change auditing.

Materials and Methods

This study, conducted in June-July 2025, used methods of theoretical modelling, experimental research, software implementation, and systematic analysis of the regulatory and legal framework. Using the JavaScript language and the Online JavaScript Compiler (Editor) – Programiz platform, a basic scheme was implemented for generating a cryptographically signed user-interface element in JSON format. To ensure controlled integrity, a Secure Hash Algorithm (SHA-256) hash function was calculated, and to confirm authenticity, the hash was signed using the Hash-based Message Authentication Code (HMAC) algorithm and a symmetric secret key. The same platform and language were also used to demonstrate an example of a Redis Pub/Sub message about cache invalidation. Theoretical foundations for ensuring UI-content authenticity in CVCLog were analysed by components – Merkle tree, principle of non-repudiation, digital signature, Access Control List (ACL), and Conflict-free Replicated Data Type (CRDT) (Badra & Borghol, 2018; Cai *et al*., 2022; Almeida, 2024). It was emphasised that SDUI directly transmitted ready-made React components through React server components.

Using the R language and the RStudio environment, a diagram of React rendering evolution (from React 16 to React 19) was constructed. The diagram visualised the chronological development of React, with major releases and key technological innovations of each version. The regulatory compliance of the CVCLog architecture was analysed for the Payment Card Industry Data Security Standard (PCI-DSS) (Agarwal *et al*., 2024), Payment Services Directive 2 (PSD2) (Christensen, 2025), and General Data Protection Regulation (GDPR) (Sienkiewicz, 2025). In addition, examples of security incidents were provided, such as the 7-Zip vulnerability in Ukraine (Girnus, 2025) and the data breach in the Episource medical company in the USA (Fadilpašić, 2025). These examples illustrated the potential consequences of insufficient data and interface protection, highlighting the necessity of cryptographic verification and auditing in the CVCLog architecture to ensure compliance with security standards such as PCI-DSS, PSD2, and GDPR.

Using the StackBlitz platform, the React library, and JavaScript, client-side Merkle proof verification for a JSON block in React was implemented. The crypto-js library was

used for cryptographic computations, performing SHA-256 hashing required for Merkle-proof generation and validation. The verification logic was implemented as a function that sequentially combined the hashes of the input JSON block with proof elements, forming a final hash compared against the signed Merkle tree root. The same platform and language were used for profiling rendering time of the App and Button components in React DevTools. To evaluate overhead from Merkle-proof verification in the client environment, Web Crypto API was used to compute SHA-256 hashes. The experiment processed 100 synthetic JSON blocks with a Merkle-proof depth of 5, for which the average integrity-verification time was measured. These blocks were selected to provide a representative sample sufficient for assessing algorithm performance under conditions close to real SDUI use. A test module was implemented to assess the interface's protective potential, analysing the number of attacks and the proportion of successful blocking, and presenting the results of PenTest scanning.

Rendering delays of UI components before and after optimisation were visualised under various network-throttling conditions. The graph was built within a canvas element using the Chart.js API and automatically initialised after the React component loaded. Synthetic pre- and post-optimisation delay values were specified for four typical networks (normal, slow3G, fast3G, Wi-Fi), represented as two data series. Additionally, cache-hit rate dynamics over time under different load conditions (low, medium, and high) were demonstrated. Cache-hit-rate simulations were performed on the StackBlitz platform using Chart.js, modelling 50 cache requests across 30-time intervals under three load conditions with base hit probabilities and random ± 5% fluctuations to simulate real environments.

For collecting interface-performance metrics in the client environment, the official web-vitals package from Google was used, implementing measurement of key Web Vitals metrics – Time to First Byte (TTFB), First Contentful Paint (FCP), Largest Contentful Paint (LCP), and Cumulative Layout Shift (CLS). Additionally, examples of input JSON requests and output blueprint files were created using the JSON Editor Online platform, along with an implemented UI-component change log in JSON format. The Service Organisation Control 2 (SOC 2) and Open Worldwide Application Security Project (OWASP) were analysed. The technical specifications of the experimental environment included an HP 250 15.6 inch G10 personal computer (Intel Core i5 processor) and Google Chrome browser version 138.0.7204.185.

Results

Architecture of a secure user interface change log

In modern distributed systems, where the user interface is generated on the server side, the authenticity, and immutability of the UI delivered to the client acquire critical importance. This is especially relevant for financial, medical, and governmental applications, where even a minor alteration of interface elements may cause data leakage, fraud, or regulatory non-compliance. To minimise the risks of manipulation at the client level or within the intermediary infrastructure (content delivery network, proxy, edge functions), an architecture with cryptographic verification of changes and subsequent audit capability is required. In response to these challenges, the CVCLog concept is introduced, providing SDUI systems with means to ensure integrity, verifiability, and immutability of interface content.

CVCLog is a formalised data structure that records every change in UI blocks transmitted from the server to the client, with a guarantee of impossibility of undetected editing or deletion. The foundation of the log consists of structured JSON packets describing the state of UI elements (buttons, forms, messages), along with cryptographic signatures and a Merkle tree of hashes ensuring the integrity of the entire sequence of changes. The log may be implemented as a temporary in-memory repository (e.g., Redis), a permanent log in a database (PostgreSQL or Append-Only Log in S3), or as a hybrid with caching of the latest changes. Each request to the interface (for example, /ui/tenantA/admin) returns not only the UI document but also a Merkle proof, allowing the client to independently verify its authenticity. This ensures non-repudiation of changes, compliance with audit requirements, and trust in the UI even within a compromised environment.

To ensure modification resistance of the user interface in SDUI, each interface block is formed as a structured JSON packet describing its appearance, behaviour, and interaction parameters. To guarantee immutability, such a packet is digitally signed on the server side, and the signed packets are organised into a Merkle tree, enabling rapid and efficient verification of the authenticity on the client side. Thus, any attempt to modify an individual interface element results in a disruption of the hash-tree structure, which is immediately detected during verification. Below is a basic example of constructing a JSON block for a button and computing its hash with a digital signature (Fig. 1).

The code shown is written in JS using the SHA-256 algorithm. This example demonstrates how an individual interface element (e.g., a payment confirmation button) can be protected by hashing and signing already at the stage of generation on the server. The computed SHA-256 hash guarantees detection of even the smallest data changes, while the HMAC signature records the source of generation. Such atomic blocks, signed at the moment of creation, serve as the building elements for forming a full CVCLog, which is subsequently used for verifying integrity and immutability on the client side. This enables SDUI systems to detect falsifications or substitutions of elements even in the event of content delivery network or proxy compromise. To better understand the protection mechanisms embedded in the CVCLog architecture, it is necessary to examine the fundamental theoretical components: Merkle trees, the principles of non-repudiation, digital signatures, and basic concepts of access and replication (Table 1).

main.js Share Run

```
1  const crypto = require('crypto');
2  const uiBlock = {
3    type: 'button',
     text: 'Confirm payment',
     action: '/submit-payment',
     style: 'primary'
   };

   const jsonData = JSON.stringify(uiBlock);
10 const hash = crypto.createHash('sha256').update(jsonData).digest('hex');
11
12 function signData(data, privateKey = 'server-private-key') {
13   return crypto.createHmac('sha256', privateKey).update(data).digest('hex');
14
15
16 const signature = signData(hash);
17
18 const signedBlock = {
19   data: uiBlock,
20   hash: hash,
21   signature: signature
22 };
23
24 console.log(signedBlock);
```

Output Clea

```
{
data:
  type: 'button',
  text: 'Confirm payment',
  action: '/submit-payment',
  style: 'primary'
}
hash: '2a517d9000a1fef2906dda92a558875a76625b342d1f09537e6e23ef0d74c954',
signature: '360f0b5b134849a4b9658566118df80533bbaf77170fa4c18225ac01e1f3c6bd'

=== Code Execution Successful ===
```

Figure 1. Generation of a signed UI block with hashing and HMAC signature based on JSON

Source: created by the authors

Table 1. Theoretical foundations of ensuring UI content authenticity in CVCLog

Component	Brief description	Example of use in CVCLog/SDUI
Merkle tree	A hierarchical hash structure ensuring integrity and non-repudiation with lower computational cost	Verification of UI JSON-block integrity via root hash and Merkle proof
Non-repudiation principle	Guarantee that the author cannot deny having signed a message, implemented through hashes and Merkle trees	The server cannot deny creation of UI blocks; the proof records the sequence
Digital signature	Cryptographic authentication mechanism using key pairs to protect against forgery	Each UI JSON block is signed by the server; the client verifies the signature
ACL	Access control lists defining user/role permissions to resources	Restricting access to UI blocks (e.g., administrative forms)
CRDT	Data enabling consistent merging of changes in distributed systems even during network failures	Replication of UI states in offline modes or on the edge without loss of consistency

Source: created by the authors based on M. Badra & R. Borghol (2018), X.-Q. Cai *et al*. (2022), P.S. Almeida (2024)

Table 1 summarises the key theoretical components that ensure reliability, authenticity, and immutability of UI content in the CVCLog architecture, highlighting efficient approaches capable of functioning even under limited computational resources or quantum threats. These components underpin the construction of reliable, scalable SDUI systems. In general, SDUI is an architectural approach in which the server defines and generates the user interface, sending the client a description of the UI in the form of structured data, usually JSON. The client, in turn, receives this data and renders the interface according to the provided instructions. This approach enables centralised UI management, ensuring consistency and flexible updates without requiring client-side application releases. Therefore, SDUI emerged as a response to the limitations of traditional architecture, where any interface modification required a new client release, slowing development and deployment of updates.

In this context, the term Backend-Driven UI is often used synonymously with SDUI, but it emphasises not only interface management but also delegation of client logic to the server, including display rules, element behaviour, and adaptation to roles, context, or A/B testing. Here, the backend serves as the single source of truth for the entire interface. It is also worth noting that in modern SDUI implementations, the server not only sends a UI description as JSON but directly streams ready-to-render React components through React Server Components, dynamically forming the UI depending on the response of large language models or real-time data. Such an approach within Backend-Driven UI allows delegating not only structure but also display logic, validation, and behavioural rules to the server, simplifying the client side and ensuring flexible interface adaptation to user context, request, or role. The technological foundation for efficient implementation of the Backend-Driven UI concept has been the development of React – particularly the introduction of server components. A brief timeline of React rendering evolution demonstrates key stages of the technology's progression, from client-side rendering in React 16 through server-side rendering and concurrent rendering to the modern server components in React 19. Figure 2 illustrates these milestones with major innovations and release years.

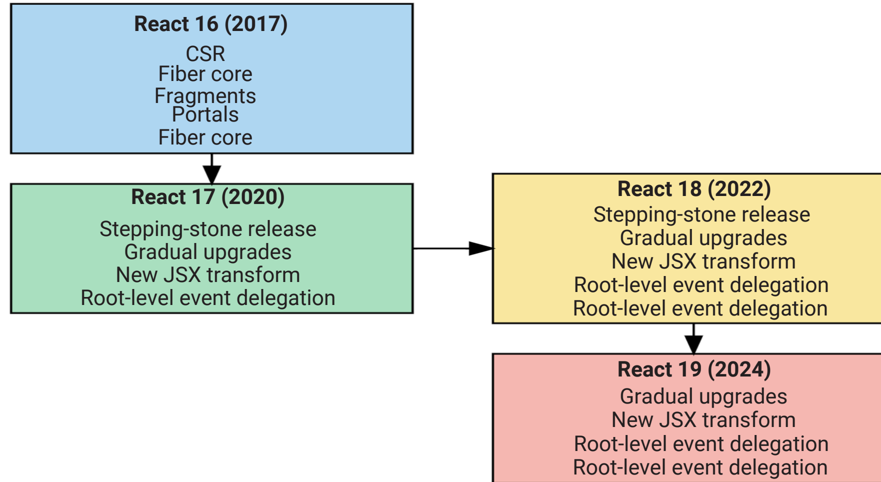

Figure 2. Evolution of React rendering: from React 16 to React 19
Note: CSR – Client-Side Rendering, JSX – JavaScript eXtensible Markup Language, SSR – Server-Side Rendering
Source: created by the authors

The presented scheme demonstrates the gradual sophistication and optimisation of React rendering approaches, enhancing interface performance, flexibility, and scalability. It is on this technological basis that CVCLog integration with the React client is realised. The integration of the log involves implementing mechanisms for verification of Merkle proofs directly on the client side. React components receive signed JSON blocks and corresponding integrity proofs, enabling real-time verification of UI authenticity and prevention of rendering falsified or modified content. This approach provides reliable interface protection even in the event of network-infrastructure compromise.

Given the gradual development of rendering technologies in React, integration of CVCLog with the client side becomes an important step in ensuring interface integrity and security. However, to implement such mechanisms effectively, compliance with regulatory requirements concerning data protection and transaction security must also be considered. Table 2 summarises the key standards influencing the architecture of the secure change log.

Table 2. Regulatory compliance of CVCLog architecture

Standard	Brief description	Impact on CVCLog architecture
PCI-DSS	Global security standard for organisations handling payment cards, emphasising endpoint and access-log protection	Ensures UI integrity through cryptographic signatures and Merkle trees; meets log-protection and client-security requirements
PSD2	EU regulation for payment services requiring strong authentication and access control	Guarantees a secure and transparent UI, minimising forgery and unauthorised-access risks
GDPR	EU regulation for personal-data protection emphasising privacy, encryption, and audit	Minimises processing of clients' personal data, ensures transparency and non-repudiation of changes, complies with security and access-control requirements

Source: created by the authors based on M.K. Agarwal *et al.* (2024), L.D. Christensen (2025), H. Sienkiewicz (2025)

These standards define requirements for protection, integrity, and transparency of interface content in the CVCLog architecture, underscoring the importance of cryptographic verification and access control for compliance with modern security norms. Meanwhile, disruption of the UI flow – illustrating insufficient interface protection – may have serious consequences, as evidenced by specific incident examples from various countries. For instance, in September 2024, a vulnerability in 7-Zip allowed attackers to bypass Windows Mark-of-the-Web protection, leading to execution of malicious code via double archiving (Girnus, 2025). This vulnerability was actively exploited in the SmokeLoader campaign targeting governmental and public organisations in Ukraine. In the United States, in 2025 the medical company Episource suffered a data breach affecting 5.4 million users due to server compromise (Fadilpašić, 2025). Attackers gained access to medical records, personal data, and insurance information.

To summarise, the CVCLog architecture is based on cryptographic verification of signed JSON blocks organised into a Merkle tree, ensuring immutability and non-repudiation of UI changes even in complex distributed systems. This approach integrates with modern SDUI and React technologies, allowing the client to verify UI-content authenticity in real-time while complying with strict regulatory requirements (PCI-DSS, PSD2, GDPR). Thus, CVCLog provides a robust protective layer against UI tampering, reducing the risks of data leakage, fraud, and security-breach penalties.

Implementation of verification mechanisms, API audit and experimental evaluation

To prevent the visualisation of compromised interface blocks in the client environment, implementation of a mechanism for verifying JSON-structure integrity using Merkle proofs is crucial. The first stage involves creating a client function that validates a Merkle proof based on the input JSON, proof tree, and signed root. Only after successful verification are the data passed to the React component. Figure 3 shows a basic implementation of this mechanism.

```
1  import React, { useEffect, useState } from 'react';
2  import sha256 from 'crypto-js/sha256';
3
4  const verifyProof = (leaf, proof, root) => {
5    let hash = sha256(leaf).toString();
6    for (const { hash: sibling, position } of proof) {
7      hash =
8        position === 'left'
9          ? sha256(sibling + hash).toString()
10         : sha256(hash + sibling).toString();
11   }
12   return hash === root;
13 };
14
15 export default function App() {
16   const [isValid, setIsValid] = useState(null);
17   const [duration, setDuration] = useState(null);
18
19   const jsonBlock = JSON.stringify({
20     component: 'Button',
21     props: { text: 'Submit', style: 'primary' },
22   });
23
24   const proof = [
25     { hash: sha256('sibling1').toString(), position: 'left' },
26     { hash: sha256('sibling2').toString(), position: 'right' },
27   ];
```


Figure 3. Fragment of client verification code of a Merkle proof for a JSON block in React
Source: created by the authors

In this program, an example of a UI component of the Button type is demonstrated, which is serialised into JSON and hashed for further integrity verification. The computation is performed by sequentially combining the hashes of the JSON block with the hashes of the proof elements according to the positions, after which the resulting value is compared with the Merkle tree root hash. In the given case, the verification passes successfully, which is reflected in the status of the React component. Additionally, the execution time of the operation on the client side is measured – 0.1 milliseconds (ms) – simulating the performance of the Predictive Prefetch mechanism. This approach allows integrity verification to be integrated directly into the client part of SDUI, ensuring protection from forgery or unauthorised modifications of interface elements before the rendering.

The next step is the implementation of reactive verification, which enables real-time tracking of changes in interface data and automatic integrity checks during updates. Unlike one-time verification before rendering, reactive verification provides continuous monitoring, which is particularly important in dynamic SDUI where changes occur constantly. It is based on observing data streams or state-change events, which makes it possible to respond promptly to potential integrity violations.

In general, for the effective integration of SDUI mechanisms into React applications, it is important to ensure the possibility of gradual loading and activation of interface components. This functionality is implemented by React's architectural capabilities – Suspense, Streaming, and Partial Hydration. Suspense makes it possible to "pause" rendering of individual components until asynchronous operations, such as obtaining a blueprint configuration from the server, are completed. In combination with server streaming, this allows the interface to be delivered in parts as soon as the components are ready, reducing the time to first render and increasing overall performance.

In turn, Partial Hydration enables activation of only those DOM parts that have dynamic behaviour, leaving static elements uninitialised until necessary. This allows flexible integration of SDUI architecture with React without complete re-initialisation of all components on the client side. The next stage is the implementation of an API-call scenario for obtaining blueprint configurations depending on the user context (Fig. 4).

```
{
  "tenant": "acme-manager",
  "role": "admin"
}
```

Figure 4. Input JSON request
Source: created by the authors

In this example, the request is sent to the endpoint GET/ui/{tenant}/{role} with corresponding parameters. It identifies the user as an administrator of the organisation, based on which the server generates the appropriate blueprint configuration of the interface. In response, the server

sends a blueprint file – a JSON structure containing a description of interface components together with metadata required for client-side integrity verification (Fig. 5).


```
{
  "components": [
    {
      "type": "Button",
      "props": {
        "label": "Create Report",
        "action": "/reports/create"
      },
      "hash": "fae124d8d9f7d3b2c5e87cb309e46c098c",
      "proof": [
        { "position": "left", "hash": "a1b2c3d4e5f6..." },
        { "position": "right", "hash": "d4e5f6a1b2c3..." }
      ]
    }
  ],
  "merkleRoot": "3c6e0b8a9c15224a8228b9a98ca1531d"
}
```


Figure 5. Output blueprint file
Source: created by the authors

In this example, the server returns a Button component with the label Create Report and the action /reports/create. The response also contains the hash of each block, the Merkle proof, and the signed root hash. Such a structure enables independent client-side verification of the received data before the rendering, which is a key principle of secure SDUI. To ensure traceability of changes in SDUI, it is necessary to implement API auditing, which records every modification of blueprint configuration – including content, structure, or access-rights changes to UI components. Such auditing is critically important in the context of access control to sensitive interface elements (i.e., PII zones) according to the requirements of SOC 2 and OWASP. For example, SOC 2 is an audit standard defining requirements for control over security, confidentiality, processing integrity, availability, and data privacy (Joodala, 2025). It requires active logging of access to UI components with confidential or personal data, control of interface-configuration changes, and the ability to reproduce the history of operations on PII zones for auditing purposes.

In turn, OWASP Top 10 is an international standard identifying the most critical web-application vulnerabilities (Li & Li, 2025). It is based on the analysis of thousands of incidents and covers threats related to data integrity, access control, API protection, and security configuration. The 2025 version of OWASP Top 10 focuses on risks associated with artificial intelligence (AI), complexity of APIs and cloud systems, software-supply-chain attacks, and the strengthening of regulatory requirements. Compliance with these recommendations is essential for ensuring SDUI security, particularly through logging, integrity verification, and access control. To implement such control, detailed logging of every interface-configuration change must be maintained, ensuring transparency and the ability to restore the change history at any time. Each time a UI file is updated on the server, the event is recorded in a change log considering the time of change, author, UI-fragment identifier, hash of the new state, and reference to the MerkleRoot. Figure 6 shows an example of a change-log entry.


```
{
  "timestamp": "2025-07-23T14:52:11Z",
  "tenant": "acme-manager",
  "role": "admin",
  "modifiedBy": "user_42",
  "changeType": "update",
  "componentId": "button_create_report",
  "newHash": "e7f8a9d1c2b3...",
  "merkleRoot": "3c6e0b8a9c15224a8228b9a98ca1531d"
}
```


Figure 6. Fragment of the UI component change log in JSON format
Source: created by the authors

In addition to logging, a mechanism for restoring the history of changes is implemented: using the API request GET /ui/history/{tenant}/{role}, previous versions of blueprint files with the hashes and timestamps can be retrieved. This allows not only forensic analysis but also restoration of a previous state in case of compromise. Such an approach to API auditing integrates integrity, observability, and rollback of changes into a single secure UI-management cycle that complies with the principles of transparency and accountability in modern information systems. For timely notification of clients about changes in UI configuration and to ensure cache validity, the Redis publish-subscribe (Pub/Sub) mechanism is used (Fig. 7).


```
[2025-07-24T10:15:32Z] Channel: ui_invalidation
Message: {
  "tenant": "acme-manager",
  "role": "admin",
  "updatedComponent": "Button",
  "newMerkleRoot": "3c6e0b8a9c15224a8228b9a98ca1531d",
  "timestamp": "2025-07-24T10:15:32Z"
}
```


Figure 7. Example of Redis Pub/Sub message about cache invalidation
Source: created by the authors

When a blueprint file is updated, the server publishes a message in a Redis channel, signalling the need to invalidate the cache of the corresponding UI block. Figure 7 illustrates an example of a Redis message about cache invalidation of a UI component. In the example provided, the server notifies about the update of the Button component for the administrator, indicating the new merkleRoot value, allowing clients to promptly delete outdated cache and request current data. To analyse rendering overhead after receiving updates, the React DevTools Profiler instrument was used (Fig. 8).

Figure 8. Profiling of rendering time of App and Button components in React DevTools

Source: created by the authors

The results showed that rendering of the App component lasted 2.7 ms, of which 1.8 ms was active updating, while the Button component was rendered for 0.4 ms (of which 0.2 ms was actual change). This indicates low overhead during UI updates in response to configuration changes, confirming the feasibility of integrating client-side verification mechanisms in dynamic SDUI applications. A complement to this analysis is the data obtained during performance modelling of client-side verification and evaluation of interface-level attack resistance, presented in Figure 9.

```
1  import React, { useEffect, useState } from 'react';
2
3  function hashJsonBlock(block) {
4    return crypto.subtle
5      .digest('SHA-256', new TextEncoder().encode(block))
6      .then((buf) =>
7        Array.from(new Uint8Array(buf))
8          .map((b) => b.toString(16).padStart(2, '0'))
9          .join('')
10     );
11 }
12
13 async function verifyMerkleProof(leafHash, proof, root) {
14   let current = leafHash;
15   for (const sibling of proof) {
16     const combined = current + sibling;
17     current = await hashJsonBlock(combined);
18   }
19   return current === root;
20 }
21
22 async function simulateExperiment(numBlocks = 100, proofDepth = 5) {
23   let totalTime = 0;
24   let pass = 0;
```


Figure 9. Fragment of React application for evaluating Merkle-proof verification overhead and results of PenTest scanning

Source: created by the authors

This program demonstrates an experimental evaluation of overhead for verifying Merkle proofs of 100 JSON-data blocks, of which 92 (92%) were successfully verified, with an average verification time of about 0.328 ms per block, indicating the algorithm's efficiency in the client environment. Also presented are the results of simulated PenTest scanning, in which 47 of 50 attack scenarios were successfully blocked, corresponding to a 94% protection level, demonstrating the potential of interface-level defence mechanisms. To better assess the effect of optimisations on interface performance, a separate analysis of UI-component rendering delays under various network conditions using network throttling was conducted, the results of which are shown in Figure 10.

The graph compares delay before and after optimisation in the client React application under different network conditions. The highest delay before optimisation was observed under slow3G – up to 850 ms – whereas after optimisation it decreased to 300 ms. Under Wi-Fi and normal-latency conditions, the delay dropped to less than 50 ms, demonstrating a significant reduction in rendering time with low network latency. The graph illustrates that optimisation mechanisms (e.g., predictive preloading, partial hydration) effectively reduce latency across all connection types, which is particularly critical for dynamic SDUI applications. However, it is equally important to maintain a high cache-reuse ratio, especially under different load conditions – this dependence is shown in Figure 11.

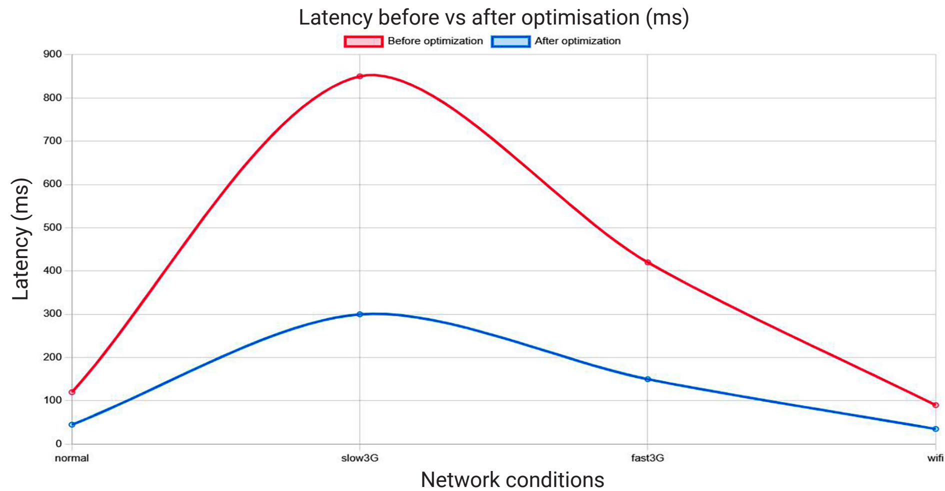

Figure 10. Graph of UI-component rendering delay before and after optimisation under various network-throttling conditions

Source: created by the authors

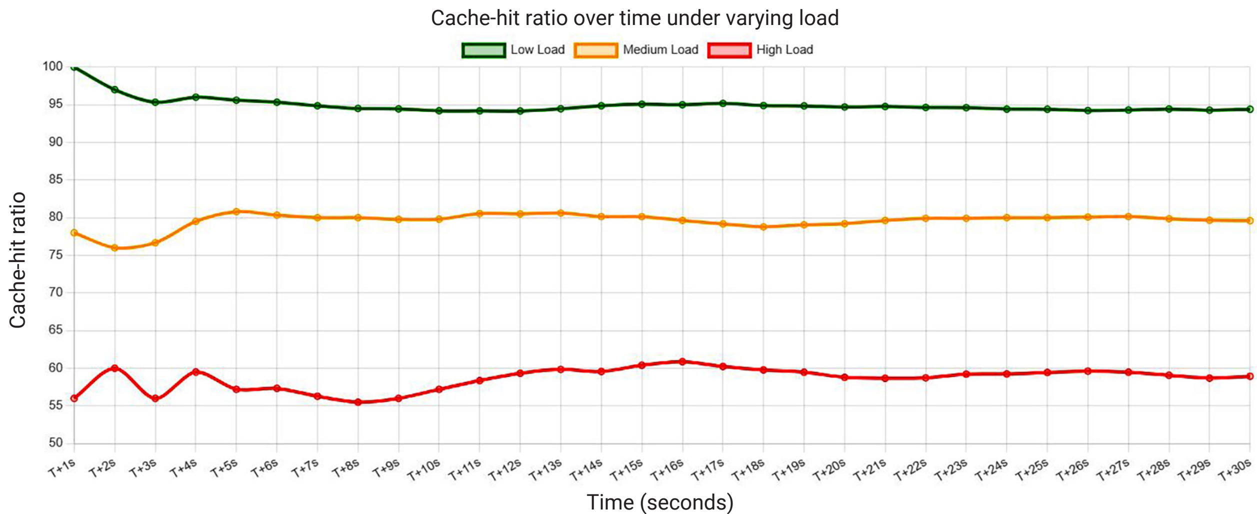

Figure 11. Graph of cache-hit rate over time under different load conditions

Source: created by the authors

The graph displays cache-hit-rate changes over a 30-second period under three load conditions (low, medium, and high), with a base hit probability defined for each (0.95, 0.80, and 0.60, respectively). Low load consistently shows the highest hit rate – from 100% initially to stable 94-95% after a few seconds. Medium load exhibits a lower level – around 77-81%, tending to stabilise near 80%. High load records the lowest cache-hit rate – between 55% and 61% – with notable variability at the start and stabilisation around 58%. Thus, the results demonstrate a clear dependence of caching efficiency on load: as load increases, the hit rate decreases, which is critically important for SDUI systems with high-frequency interface updates.

Additionally, Web Vitals are important – standardised metrics reflecting real user experience when interacting with an interface. For example, TFB is the time between a request and receipt of the first byte of the response; FCP is the time until the browser displays the first visual DOM element; LCP is the rendering time of the largest visible element on the screen. For analytics, quantiles p50 (median) and p95 (worst 5% scenario) are used. The measured values before and after optimisation are shown in Figure 12.

📊 **Web Vitals Metrics**

- TTFB: 558.10 ms
- FCP: 648.00 ms
- LCP: 764.00 ms
- CLS: 0.00 ms

Data is collected automatically via web-vitals API.

Figure 12. Measurement of Web Vitals metrics before/after cache and Predictive Prefetch implementation

Source: created by the authors

After implementing caching of blueprint configurations and the Predictive Prefetch algorithm, improvements in FCP and LCP were observed due to reduced network-loading delays. Specifically, p95 LCP metrics decreased by tens of milliseconds, which is particularly noticeable during repeated component loads. The results confirm the expediency of integrating predictive preloading into SDUI applications to improve performance.

Thus, implementation of interface-integrity verification based on Merkle proofs ensures tamper-resistance of SDUI applications, preventing undetected modifications of interface blocks even in a compromised environment. The introduced optimisations – including caching, Predictive Prefetch, and partial hydration – significantly reduce rendering delays and improve key Web Vitals metrics, especially under limited network resources. A comprehensive approach combining client-side proof verification, API auditing, cache control, and reactive verification forms a robust architecture that meets regulatory-standard requirements and ensures a high level of security, transparency, and resistance to interface-level attacks.

Discussion

In this work, client verification of Merkle proofs in SDUI was implemented, ensuring protection of the interface from unauthorised modifications with a verification delay of 0.328 ms, even under high load conditions. In turn, M. Ethan (2025) proposed a frontend-driven backpressure handling model in real APIs, which improves latency, memory control, and interface stability under load. Hence, both approaches aim to ensure the resilience of the client side to overload, but the current system additionally guarantees cryptographic data integrity.

In the conducted study, an SDUI model was implemented with client-side integrity verification of signed JSON packages using Merkle proofs, which prevents interface configuration tampering and ensures immutability similar to blockchain solutions. Conversely, in the work of I. Shahzad *et al.* (2025), an Internet of Things (IoT) system using a trusted blockchain for secure sensor management and data recording with immutability and controlled access guarantees was presented. Both approaches are aimed at enhancing trust in dynamic systems through the use of immutable data structures; however, the current solution focuses on verifying interface data on the client side before rendering.

Within this study, a mechanism for verifying interface JSON structures in SDUI through Merkle proofs was created, ensuring auditability of dynamic UI without loss of performance. On the other hand, G. Sharma (2025) proposed an architecture with a blockchain embedded in the operating system kernel for immutable logging of AI decisions, achieving 100% accuracy in forgery detection and reducing audit preparation time to real-time. Both approaches align in the pursuit to ensure transparency and compliance with audit requirements through record immutability, although the current solution achieves this at the client-interface level with minimal delays, whereas the mentioned model implements similar properties at the OS-kernel level, accompanied by higher system costs.

The results proposed the CVCLog model for SDUI, ensuring immutability, authenticity, and detailed auditing of UI components in real-time without performance loss. In comparison, S. Fugkeaw *et al.* (2025) presented the scheme Efficient and Verifiable Searchable Encryption with Boolean Search, which ensures integrity and searchable accessibility of cloud logs through a combination of indexing, blockchain ACL, and Merkle-root verification. Both approaches are consistent in striving to guarantee controlled immutability of data; however, the current system is oriented specifically at the UI level and allows detection of modifications even before rendering, which is critical in a real-time context.

Analysis of the requirements of PCI-DSS, PSD2, GDPR, SOC2, and OWASP standards for interface integrity, change logs, and security of client components enabled the formation of a cryptographically verifiable UI architecture that complies with regulatory requirements. In turn, P.R. Vennamaneni (2025) examined the compliance of AI systems in the financial sector with PCI-DSS and IEC 62304 standards, focusing on transaction data protection, decision-making transparency, and cloud infrastructure. Thus, the results of the presented study confirm the expediency of adhering to standards in critical digital systems, while the current work demonstrates a concrete implementation of this compliance at the client-interface level.

This work examined the evolution of React rendering (versions 16-18), integration of Server Components, Suspense, and Partial Hydration to ensure continuous rendering during client verification of Merkle proofs, enabling UI integrity control prior to its display. In turn, S. Wagh *et al.* (2025) presented React-Nex – a modular component library with AI-driven code generation support via Retrieval-Augmented Generation and vector embeddings, accelerating the creation and configuration of interface elements. Both approaches are aimed at improving the efficiency of React-application development; however, the current solution focuses on interface protection and verification, whereas React-Nex is oriented towards automation and flexibility in UI-component generation.

The proposed CVCLog architecture for SDUI ensures cryptographic integrity of interface data and detailed real-time audit of changes, including through client-side verification of Merkle proofs prior to rendering. Regarding the work of N. Jose (2025), it demonstrated the advantages of an event-driven architecture for real-time synchronisation of inventory data in retail systems, emphasising consistency and event-processing speed. Both approaches align in the pursuit of building reactive, scalable, and transparent real-time systems; however, the current CVCLog architecture additionally addresses the task of verifying data authenticity at the client level, which is critical for secure interactions in distributed interfaces.

The conducted study proposed the CVCLog architecture for SDUI, in which configuration JSON interface blocks

are organised as a Merkle tree, and integrity verification is performed client-side before rendering by computing and comparing Merkle proofs with the signed root hash. Similarly, A. Odeh & A. Abu Taleb (2025) proposed a Blockchain-Enhanced Trust and Access Control for IoT Security model, which uses Merkle trees, blockchain, and federated learning for decentralised access control and data integrity verification in IoT environments. Both approaches aim to strengthen trust in dynamic, distributed systems through built-in authenticity verification and decentralised control mechanisms, although the current solution implements these properties directly in the UI layer, enabling interactive protection of the user interface without performance loss.

Additionally, A. Osilaja *et al.* (2024) justified the use of blockchain for building secure and transparent software architectures, focusing on data immutability, decentralised access control, and cryptographic verification. In alignment with this approach, the current CVCLog architecture is also oriented towards ensuring trust in dynamic systems but implements these principles directly at the user-interface level. The distinction lies in the fact that the proposed solution enables interactive verification of UI-component authenticity in real-time, whereas the mentioned authors focus on general aspects of software security.

The current CVCLog architecture in the SDUI environment ensured verification of interface JSON-data authenticity before rendering, with an average Merkle-proof verification time of 0.328 ms and attack-detection accuracy of up to 94%, without affecting interface performance. Meanwhile, A.A. Sathio *et al.* (2025) proposed the ClusterPioneer model for trusted blockchain, which reduces communication load by 60%, provides verification within 150 ms, and achieves 95% accuracy in attacker detection. Hence, the results of the mentioned work confirm the expediency of using Merkle structures and decentralised verification to ensure data integrity in critical digital systems, while the current solution demonstrates the application of these principles at the client UI level with minimal delays.

The conducted experiments confirmed that integration of cryptographic verification with React Suspense and Partial Hydration ensures stable UI rendering with minimal impact on response time (2.7 ms for the main component) even under high loads. Conversely, Y. Chavan *et al.* (2025) presented Nexify – a scalable real-time server for online communities providing low latency, modularity, role-based access control, and end-to-end encryption, as well as incorporating blockchain identification and AI moderation. Thus, both approaches are oriented towards building secure, scalable, and reactive systems, while the current solution complements the Nexify framework with the ability to verify interface authenticity prior to its display – a critical element of trust in dynamic environments.

Overall, the conducted study focuses on cryptographic verification of UI-data integrity in real-time with minimal delays and high change-detection accuracy at the client level. In turn, the work of B. Ganji *et al.* (2024) focused on formal verification of distributed streaming systems to prevent process deadlocks and errors already at the design stage. Therefore, both approaches complement each other, combining data protection with architecture correctness assurance in complex real-world systems.

To summarise, this work proposed the CVCLog architecture for SDUI, which implements client-side verification of interface JSON-data integrity using Merkle proofs with low latency and high change-detection accuracy. This approach ensures UI protection from unauthorised modifications before rendering, supports detailed real-time auditing, and complies with leading security and audit standards. Thus, the study not only complements existing solutions for load control and data security but also advances practical methods for ensuring trust in dynamic client interfaces in distributed web systems.

Conclusions

The conducted study confirmed the effectiveness of the CVCLog architecture as a reliable mechanism for ensuring integrity and immutability of UI in SDUI systems. The obtained results emphasised that the use of cryptographically signed JSON blocks and Merkle trees allows detection of any unauthorised changes on the client side even under network infrastructure compromise conditions. Implementation of a change log with non-repudiation support guarantees compliance with high regulatory standards such as PCI-DSS, PSD2, and GDPR. Furthermore, integration with the React client ensures verification of UI authenticity in real-time without significant impact on performance.

It was also established that client verification of Merkle proofs in a React application is performed with an average time of 0.1 ms per block, confirming its efficiency for integration into dynamic SDUI. Reactive verification ensures continuous control of UI changes in real-time, reducing the risks of content tampering. Rendering profiling showed that updating the main App component takes 2.7 ms, while an individual Button component requires 0.4 ms, indicating low overhead for client-side verification. In turn, experimental verification of JSON blocks revealed 92% successful verifications, with an average time of 0.328 ms per block. Meanwhile, PenTest scanning results demonstrated 94% attack blocking at the UI level, and cache hit rate in stable mode reached 94-95% under low load and decreased to 58% under high load. Implementation of optimisations improved key Web Vitals metrics: p95 LCP decreased by tens of milliseconds, enhancing response time during repeat loads.

The study's limitations lie in the dependency of verification and caching performance on client hardware resources and network quality, which may restrict the applicability of the proposed mechanisms in low-power or highly constrained environments, as well as in the complexity of scaling reactive verification under very high UI update frequency. For further development, it is recommended to optimise verification algorithms using hardware acceleration and to design adaptive caching strategies considering load variability and network conditions. An important direction is integration of machine-learning mechanisms for

predicting UI changes and preloading the most probable components, which would further reduce rendering latency.

Acknowledgements
None.

Funding
The study was not funded.

Conflict of Interest
None.

References
[1] Agarwal, M.K., Sarden, D., Ramesh, S., & Singh, R. (2024). Endpoint controls through a lens of PCI DSS. In M. Gupta, R. Singh, J. Walp & R. Sharman (Eds.), *Advances in enterprise technology risk assessment* (pp. 245-282). London: IGI Global. doi: 10.4018/979-8-3693-4211-4.ch009.
[2] Almeida, P.S. (2024). Approaches to conflict-free replicated data types. *ACM Computing Surveys*, 57(2), article number 51. doi: 10.1145/3695249.
[3] Azhar, H.B., Butt, K.K., Awan, N.U., & Irshad, O. (2025). Quantum-resistant merkle trees enhancing data integrity with post-quantum cryptography and zero-knowledge proof. *Journal of Computing & Biomedical Informatics*, 8(2). doi: 10.56979/802/2025.
[4] Badra, M., & Borghol, R. (2018). Long-term integrity and non-repudiation protocol for multiple entities. *Sustainable Cities and Society*, 40, 189-193. doi: 10.1016/j.scs.2017.11.023.
[5] Cai, X.-Q., Wang, T.-Y., Wei, C.-Y., & Gao, F. (2022). Cryptanalysis of quantum digital signature for the access control of sensitive data. *Physica A: Statistical Mechanics and its Applications*, 593, article number 126949. doi: 10.1016/j.physa.2022.126949.
[6] Chavan, Y., Jadhav, A., Kulkarni, S., Malpure, S., & Mandal, S. (2025). Nexify: A scalable and secure community server for real-time communication. *International Journal of Advanced Research in Science Communication and Technology*, 5(4), 547-551. doi: 10.48175/IJARSCT-25172.
[7] Christensen, L.D. (2025). Financial fraud and the PSD2. In L.D. Christensen (Ed.), *EU payment services: Regulation and innovation* (pp. 145-181). Oxford: Oxford University Press. doi: 10.1093/9780198949084.003.0006.
[8] Du, P., Liu, Y., Li, Y., & Yin, H. (2022). EthMB+: A tamper-proof data query model based on b+ tree and Merkle tree. In Y. Sun, L. Cai, W. Wang, X. Song & Z. Lu (Eds.), *Blockchain technology and application* (pp. 49-59). Singapore: Springer. doi: 10.1007/978-981-19-8877-6_4.
[9] Ethan, M. (2025). *Frontend-driven backpressure handling for real-time APIs*. Retrieved from https://www.researchgate.net/publication/393981918_Frontend-Driven_Backpressure_Handling_for_Real-Time_APIs.
[10] Fadilpašić, S. (2025). *Major breach at medical billing giant sees data on 5.4 million users stolen – here's what we know*. Retrieved from https://www.techradar.com/pro/security/major-breach-at-medical-billing-giant-sees-data-on-5-4-million-users-stolen.
[11] Fugkeaw, S., Deevijit, J., Ueasathitwong, P., & Thanyasukpaisal, T. (2025). EVSEB: Efficient and verifiable searchable encryption with boolean search for encrypted cloud logs. *IEEE Access*, 99, 101177-101195. doi: 10.1109/ACCESS.2025.3577466.
[12] Ganji, B., Rezaee, A., Adabi, S., & Movaghar, A. (2024). Model verification of real-time and distributed stream processing architecture. *Computing*, 107(1), article number 17. doi: 10.1007/s00607-024-01384-w.
[13] Girnus, P. (2025). *CVE-2025-0411: Ukrainian organizations targeted in zero-day campaign and homoglyph attacks*. Retrieved from https://www.trendmicro.com/en_us/research/25/a/cve-2025-0411-ukrainian-organizations-targeted.html.
[14] Havatiuk, M., & Saiapina, I. (2025). Improved method of targeted user interface updates for enhancing the efficiency of web applications based on reactive streams and virtual DOM. *Technical Engineering*, 95(1), 259-265. doi: 10.26642/ten-2025-1(95)-259-265.
[15] Joodala, A. (2025). A cloud-native approach to SOC 2, HIPAA, and GDPR compliance using AWS microservices. *International Journal of Innovative Research in Engineering & Multidisciplinary Physical Sciences*, 13(3). doi: 10.37082/IJIRMPS.v13.i3.232605.
[16] Jose, N. (2025). Event-driven architecture in retail: Real-time inventory synchronization for omnichannel retail. *International Journal of Computing and Engineering*, 7(16), 13-23. doi: 10.47941/ijce.3014.
[17] Kuznetsov, O., Frontoni, E., Kuznetsova, K., & Arnesano, M. (2025). Optimizing Merkle proof size through path length analysis: A probabilistic framework for efficient blockchain state verification. *Future Internet*, 17(2), article number 72. doi: 10.3390/fi17020072.
[18] Li, J., & Li, H. (2025). Evolution of application security based on OWASP top 10 and CWE/SANS top 25 with predictions for the 2025 OWASP top 10. In *8th International conference on inventive computation technologies* (pp. 1178-1183). Kirtipur: IEEE. doi: 10.1109/ICICT64420.2025.11004742.
[19] Odeh, A., & Abu Taleb, A. (2025). Federated learning and blockchain framework for scalable and secure IoT access control. *Computers, Materials & Continua*, 84(1), 447-461. doi: 10.32604/cmc.2025.065426.

[20] Orynchak, A., Kuzmenko, O., & Svintsytska, O. (2024). Real-time threat detection with javascript: Monitoring and response mechanisms. *Technical Engineering*, 93(1), 201-210. doi: 10.26642/ten-2024-1(93)-201-210.

[21] Osilaja, A., Raheem, A., & Edmund, E. (2024). Enhancing software security with blockchain integration for decentralized and tamper-proof application architectures. *World Journal of Advanced Research and Reviews*, 24(3), 2750-2767. doi: 10.30574/wjarr.2024.24.3.3977.

[22] Patel, O. (2022). Merkle proof verification for zero knowledge transaction validation. *International Journal of All Research Education & Scientific Methods*, 10(5), 3533-3547.

[23] Ridhorkar, S., & Mishra, S.S. (2024). Implementing quantum resistant algorithm in blockchain-based applications. *International Journal of Advanced Research in Science Communication and Technology*, 4(7), 650-659. doi: 10.48175/IJARSCT-17899.

[24] Rubel, M.T., Emran, A.K., Islam, M.K., Nayem, M.A., & Hasan, S. (2025). From ledger to ledgerless: Evaluating blockchain-driven real-time financial reconciliation in U.S. public companies. *International Journal for Multidisciplinary Research*, 7(4). doi: 10.36948/ijfmr.2025.v07i04.49709.

[25] Sathio, A.A., Rind, M.M., & Awan, S.A. (2025). ClusterPioneer voting: A scalable and energy-efficient consensus mechanism for permissioned-blockchain (DeFi) system. *Research Square*. doi: 10.21203/rs.3.rs-7099560/v1.

[26] Shahzad, I., Maqsood, M.W., Latif, S., & Ijaz, H.M. (2025). Decentralized IoT-based architectures for tamper-proof agricultural sensor networks: Ensuring end-to-end data integrity and transparent governance. *Kashf Journal of Multidisciplinary Research*, 2(5), 39-55. doi: 10.71146/kjmr442.

[27] Sharma, G. (2025). Kernel-embedded blockchain architecture for transparent AI decision auditing. *Journal of Information Systems Engineering & Management*, 10(47), 183-205. doi: 10.52783/jisem.v10i47s.9246.

[28] Shport, L. (2025). Enhancing the security of interbank payments with, a comprehensive cryptographic architecture. *Information Technology and Society*, 16(1), 276-280. doi: 10.32689/maup.it.2025.1.36.

[29] Sienkiewicz, H. (2025). *Article cybersecurity impacts of the EU GDPR*. Retrieved from https://www.researchgate.net/publication/393802678_Article_Cybersecurity_Impacts_of_the_EU_GDPR.

[30] Undirwadkar, A.J. (2025). The rise of server-driven UI: A paradigm shift in mobile app development. *World Journal of Advanced Engineering Technology and Sciences*, 15(2), 55-61. doi: 10.30574/wjaets.2025.15.2.0538.

[31] Vennamaneni, P.R. (2025). Building compliance-driven AI systems: Navigating IEC 62304 and PCI-DSS constraints. *International Journal of Networks and Security*, 5(1), 62-90. doi: 10.55640/ijns-05-01-06.

[32] Wagh, S., Vadhel, S., Tiwari, R., Bidaye, V., & Kachare, A. (2025). React-Nex – a modular component library with AI-driven code generation. *International Journal of Scientific Research in Engineering and Management*, 9(4), 1-9. doi: 10.55041/IJSREM44477.

Тампер-резистентна архітектура Server-Driven UI з верифікацією Merkle-доказів у реальному часі

Владислав Ананченко

Аспірант
Міжнародний економіко-гуманітарний університет імені академіка Степана Дем'янчука
33000, вул. Степана Дем'янчука, 4, м. Рівне, Україна
https://orcid.org/0009-0004-8963-775X

Юрій Лотюк

Кандидат педагогічних наук, доцент
Міжнародний економіко-гуманітарний університет імені академіка Степана Дем'янчука
33000, вул. Степана Дем'янчука, 4, м. Рівне, Україна
https://orcid.org/0000-0001-6696-5583

Анотація. Системи інтерфейсу користувача, що керуються сервером, потребують захисту від несанкціонованих змін для забезпечення цілісності і безпеки даних, що відображаються. Мета цього дослідження полягала у розробці криптографічно перевірного журналу змін інтерфейсу користувача для систем із Server-Driven User Interface. У межах дослідження застосовано методи теоретичного моделювання, експериментального тестування, програмної реалізації та аналізу нормативної бази для розробки, верифікації та оцінки криптографічного журналу змін у середовищі клієнтського інтерфейсу. Основні результати показали, що застосування підписаних структурованих блоків інтерфейсу із хешуванням і цифровим підписом забезпечує неможливість непомітної модифікації інтерфейсу на клієнтській стороні. Побудова журналу змін на основі дерева хешів гарантує достовірність, незмінність і криптографічну перевірку кожного елементу інтерфейсу навіть у складних розподілених умовах. Інтеграція з новітніми механізмами рендерингу React дозволяє здійснювати перевірку достовірності інтерфейсу в реальному часі, забезпечуючи відповідність вимогам міжнародних стандартів захисту персональних даних і безпеки транзакцій. Крім того, результати показали, що клієнтська перевірка Merkle-доказів для блоків у React дозволяє виявити модифікації до моменту рендерингу, із середнім часом верифікації 0,328 мілісекунди на блок. Аудит змін blueprint-файлів і система публікації-підписки забезпечили відстежуваність і актуальність даних, тоді як рендеринг компонентів після оновлень тривав лише 2,7 мілісекунди для основного компонента і 0,4 мілісекунди для кнопки. Експерименти підтвердили досягнення 94 % рівня блокування атак, зниження затримок рендерингу (з 850 до 300 мілісекунд у повільній мережі) та підтримку кеш-хітрейту на рівні 94–95 % при низькому навантаженні, що разом із покращенням ключових показників взаємодії з інтерфейсом демонструє ефективність запропонованої архітектури. Отримані результати можуть бути використані розробниками критичних вебзастосунків для впровадження захищених інтерфейсів, що перевіряють цілісність у реальному часі та відповідають міжнародним вимогам безпеки

Ключові слова: криптографічно перевірний журнал змін; принцип незаперечності; еволюція React; оцінка накладних витрат; кеш-хітрейт