# Enhancing Security for JSON-RPC 2.0: Introducing the Secure JSON-RPC Message (SJRM) Wrapper Protocol

**By: Shalak More**

## Executive Summary

The escalating complexity of modern software architectures, particularly the widespread adoption of distributed systems, microservices, and multi-agent platforms, has amplified the criticality of secure and verifiable inter-service communication.1 Distributed systems inherently introduce unique security challenges, expanding the attack surface and increasing the complexity of managing security configurations across diverse network environments.1 While JSON-RPC 2.0 has gained significant traction as a lightweight and efficient protocol for remote procedure calls, its fundamental design omits crucial per-message security mechanisms for integrity, authentication, and replay protection. This inherent vulnerability exposes distributed applications to a spectrum of debilitating risks, including insidious message tampering, unauthorized command injection, and persistent replay attacks, especially within network environments where reliance solely on transport-layer security (e.g., TLS/SSL) proves insufficient or compromised.4 The increasing adoption of distributed systems directly amplifies the criticality of message-level security, as traditional perimeter-based security or sole reliance on TLS becomes inadequate. This is a direct consequence of the network complexity and diverse technologies challenges identified in distributed systems. When messages are decrypted, for instance, at an API Gateway or within a supposedly trusted internal network segment, the individual message itself becomes vulnerable. The sheer volume and intricate nature of these interconnections mean that a breach in one part of the system can easily propagate if messages themselves are not secured at a granular level. Therefore, the architectural shift towards distributed systems necessitates a corresponding shift in security focus to the message layer.

This whitepaper comprehensively introduces the **Secure JSON-RPC Message (SJRM) Wrapper Protocol**, a pioneering solution engineered to infuse standard JSON-RPC 2.0 communications with essential cryptographic security directly at the application message layer. SJRM transparently encapsulates conventional JSON-RPC messages within a robust security envelope, integrating well-established cryptographic primitives. Key features include: cryptographically assured message integrity and authenticity verification via Hash-based Message Authentication Codes (HMAC); sophisticated replay protection achieved through the diligent use of unique, cryptographically strong nonces and stringent, time-sensitive timestamps 7; and robust sender authentication facilitated by JSON Web Tokens (JWT), which also plays a pivotal role in secure key derivation.11 The protocol's design further incorporates an extensible version negotiation mechanism to ensure future adaptability. Conceptualized to be both lightweight in its overhead and highly integrable into existing JSON-RPC deployments, SJRM empowers developers and architects to construct high-performance, inherently verifiable, and resilient distributed applications. The design of SJRM, by being lightweight and integrable, suggests a strategic approach to security that balances robust protection with operational agility. This balance is crucial for modern, rapidly evolving microservice environments, as it contrasts with heavy-handed security solutions that might impede development velocity or introduce significant performance overhead, which are common concerns in microservices architectures.13 This document meticulously details the design principles, core cryptographic components, operational flow, and significant security benefits of the SJRM protocol, substantiating its capacity to effectively mitigate critical security vulnerabilities prevalent in contemporary distributed architectures.

## 1. Introduction

The paradigm shift towards highly scalable and agile distributed systems, encompassing microservices, serverless computing, and intricate multi-agent systems, has profoundly reshaped modern software development.1 In this interconnected landscape, the efficacy and security of inter-service communication protocols are paramount. JSON-RPC 2.0, lauded for its simplicity, language agnosticism, and low network overhead, has become a favored standard for facilitating remote procedure calls between decoupled services.18 It provides a straightforward and efficient method for defining and executing remote methods, contributing significantly to the agility and responsiveness of modern applications. This protocol optimizes data serialization with JSON structures, yielding smaller message sizes compared to XML, which can improve performance by over 30% in some scenarios.18 Recent surveys indicate that 67% of organizations leveraging distributed styles report improvements in system scalability and responsiveness post-adoption of lightweight protocols like JSON-RPC, and 70% of developers prefer it for request-response interactions due to its straightforward interface and ability to support batch requests.18

However, the streamlined nature of JSON-RPC 2.0, by design, focuses solely on the serialization and deserialization of messages, omitting any intrinsic mechanisms for message-level security.4 While the use of transport-layer security protocols such as TLS/SSL is a common practice for encrypting the communication channel and authenticating endpoints, this approach offers protection primarily during transit.4 Once messages are decrypted at a service endpoint, or if an internal network segment is compromised, the individual JSON-RPC messages themselves remain vulnerable to manipulation and exploitation.4 The design of JSON-RPC 2.0 prioritizes simplicity and low network overhead, which, while beneficial for performance, directly leads to the omission of intrinsic mechanisms for message-level security. This highlights a fundamental design trade-off that the Secure JSON-RPC Message (SJRM) protocol aims to resolve. The core nature of JSON-RPC as a lightweight Remote Procedure Call (RPC) protocol means it deliberately offloads security concerns to the transport layer. This approach is efficient for basic communication but creates a significant vulnerability at the application layer. SJRM's value proposition lies precisely in adding this missing layer without negating the performance benefits, effectively addressing the "scalability and performance" challenges often encountered in distributed systems security.1

This fundamental security gap exposes applications to a spectrum of critical threats, including: unauthorized alteration of message content (tampering), masquerading as legitimate services (impersonation), and the re-execution of valid but old commands (replay attacks).27 Such vulnerabilities can severely compromise data integrity, system reliability, and the overall security posture of a distributed application. The "end-to-end principle" in system design suggests that application-specific features like reliability and security should be implemented in the communicating end nodes, rather than solely within the network itself.5 For security, this implies that while network-level encryption (such as TLS) can act as a firewall to prevent information leakage, it cannot fully satisfy application-level authentication and protection requirements. The application itself must perform end-to-end checks to ensure data integrity and authenticity, as data becomes vulnerable once it passes into the target node and is fanned out to the application. The explicit invocation of the "end-to-end argument" is crucial as it shifts the security mindset from "network perimeter protection is enough" to "application-level validation is non-negotiable for true security." This implies that even with robust network infrastructure, application developers must implement their own security checks, making solutions like SJRM indispensable for maintaining trust across distributed components, especially in complex, multi-hop environments where TLS tunnels might terminate before the final processing. For example, if a message passes through an API gateway that terminates TLS, the message content becomes "clear" at that point. If the gateway is compromised, or if the message is then routed internally to another service over an unencrypted internal network segment, it is vulnerable. The end-to-end principle argues that the application itself must be responsible for validating the message's integrity and origin, as it is the ultimate consumer of the data. SJRM provides this capability, ensuring that even if intermediate hops are compromised, the final application can still verify the message's authenticity and integrity.

This whitepaper introduces the **Secure JSON-RPC Message (SJRM) Wrapper Protocol**, a purpose-built solution engineered to address this critical security void. SJRM augments JSON-RPC 2.0 by introducing a transparent, cryptographically-secured envelope around each message, ensuring its integrity, authenticity, and freshness. The protocol leverages established cryptographic primitives: Hash-based Message Authentication Codes (HMAC) for data integrity and authenticity verification; unique, unpredictably generated nonces and tightly controlled timestamps for robust replay attack prevention; and JSON Web Tokens (JWT) for secure sender authentication and facilitating dynamic key derivation. The objective of SJRM is to provide a comprehensive, lightweight, and easily integrable security layer that empowers developers to build inherently secure, verifiable, and resilient distributed applications atop existing JSON-RPC infrastructure without requiring sweeping architectural overhauls. This document will meticulously explore the security challenges inherent in unadorned JSON-RPC communications, detail the precise design and operational mechanisms of the SJRM protocol, and articulate the compelling security and operational benefits it confers upon modern distributed systems.

## 2. The Challenge: Securing JSON-RPC 2.0 Communications

JSON-RPC 2.0 defines a simple, stateless, and efficient protocol for remote procedure calls, often relying on HTTP or WebSockets as its transport layer. A typical JSON-RPC message consists of a method name, params, and an optional id for correlation. While this design prioritizes efficiency and ease of implementation, it fundamentally leaves the individual message unprotected at the application layer. This architectural choice renders JSON-RPC communications susceptible to several critical vulnerabilities in complex, distributed environments where granular message integrity and authentication are paramount.

### 2.1 Message Tampering

Without a robust mechanism to verify the integrity of a message's content, an attacker who gains access to the communication channel (e.g., via a compromised internal network, a Man-in-the-Middle (MITM) attack, or a rogue internal service) can surreptitiously alter the parameters of a JSON-RPC request or falsify the results in a response.30 This can lead to unauthorized operations, data corruption, or misdirection of application flow. For instance, in a system managing financial transactions, an attacker could change the amount of a transfer or redirect funds to an unauthorized account without detection.30 The lack of a cryptographic checksum directly embedded within the message makes such alterations indistinguishable from legitimate messages once they have traversed the transport layer.4 Parameter tampering attacks specifically target web application parameters found in URL Query Strings, cookies, HTTP headers, and hidden HTML form fields, leading to unauthorized access or manipulation of business logic.30 Similarly, RPC attacks can exploit vulnerabilities to execute malicious code or escalate privileges on remote systems.31

The vulnerability extends beyond external attacks; a rogue internal service or a compromised internal network can also facilitate tampering.1 This highlights the shift from perimeter-centric security to a zero-trust model where internal communications also require validation. In a microservices architecture, services communicate extensively internally. If an attacker breaches the perimeter or compromises an internal service, they gain a foothold within the "trusted" network. Without message-level integrity checks, they can then inject or alter messages between internal services, bypassing any external TLS or firewall protections. This emphasizes that security must be pervasive, not just at the edge. Furthermore, the lack of an "indistinguishable" alteration points to a critical failure in auditability and forensic analysis. If a tampered message cannot be differentiated from a legitimate one, it becomes impossible to trace the source of data corruption or unauthorized actions, severely impacting incident response and compliance.35 In regulated industries like finance or healthcare, maintaining an immutable audit trail is paramount. If message tampering is undetectable at the application layer, it creates a "blind spot" in the audit log, making it difficult to recover from an attack and to prove compliance with data integrity regulations.

### 2.2 Impersonation and Unauthorized Access

Standard JSON-RPC does not provide intrinsic mechanisms for the message recipient to cryptographically verify the true identity of the message sender at the application layer.27 This vulnerability allows an attacker to forge JSON-RPC messages, masquerading as a legitimate service or authorized user, and send unauthorized commands. In a microservice architecture where services communicate directly peer-to-peer or via an internal bus, this can lead to privilege escalation or lateral movement within the system if initial perimeter defenses are breached.1 The ability to accurately identify and trust the source of each message is fundamental to maintaining authorization policies and access control within a distributed environment. The OWASP API Security Top 10 lists "Broken Authentication" and "Broken Object Level Authorization" as critical risks, where attackers impersonate users or manipulate object references.29 API impersonation attacks are a significant concern, especially when targeting unprotected devices, which serve as easy entry points for malicious actors to infiltrate networks and manipulate accounts.32

The strong independence of APIs and designers' poor considerations in the implementation of different API functions can lead to incomplete authentication logic and unauthorized access.28 This highlights a common pitfall in distributed system development where granular security is overlooked. In a large microservice system, different teams might develop services with varying levels of security maturity or inconsistent authentication practices. This creates a fragmented security posture. If one service has weak authentication, it becomes a pivot point for an attacker to impersonate it and then access other, potentially more secure, services that implicitly trust internal communications. The vulnerability of internal communications to impersonation reinforces the "zero-trust" security model, where no entity, internal or external, is inherently trusted. Every message's sender identity must be explicitly verified, regardless of its origin within the network.36 The traditional "fortress and moat" security model (strong perimeter, weak interior) is inadequate for microservices. If an attacker gains access to any internal service, they can leverage the lack of internal message authentication to move laterally and escalate privileges. A zero-trust approach, where every service interaction is authenticated, is the only way to mitigate this.

### 2.3 Replay Attacks

A replay attack involves an attacker intercepting a legitimate JSON-RPC message and subsequently re-transmitting it to the receiver at a later time.33 Even if the attacker cannot decipher or alter the message's content, replaying it can induce unintended or malicious side effects. Examples include duplicating financial transactions, re-executing sensitive commands (e.g., system restarts, data deletions), or causing resource exhaustion through repeated legitimate requests.33 Without unique, time-sensitive identifiers embedded within each message, a recipient cannot distinguish a fresh, valid message from an old, replayed one, making conventional JSON-RPC highly susceptible to this form of attack.7 Replay attacks are less demanding in terms of system knowledge and computing resources compared to other integrity attacks, making them easily implementable and stealthy (e.g., bypassing residue-based detectors).33 The impact can range from bounded estimation errors in stable systems to diverging estimation errors in unstable systems.33

Replay attacks are less demanding in terms of system knowledge and computing resources, which makes them a particularly attractive and prevalent threat in distributed systems. Attackers often prefer methods that are easy to execute and require minimal effort. The simplicity of capturing and re-sending a message, especially if it bypasses complex detection mechanisms, means that replay attacks represent a high-return, low-cost attack vector. This underscores the necessity for robust, built-in replay protection. The phrase "valid but old commands" highlights a critical temporal vulnerability. Systems that do not enforce freshness checks are susceptible to actions that were legitimate at one point but become malicious or disruptive when re-executed out of context or at a later time. This can severely impact system idempotency and state consistency.33 While many operations in distributed systems are designed to be idempotent (producing the same result regardless of how many times they are executed), a replay attack can still cause resource exhaustion or reveal sensitive information even if the core operation is idempotent. For non-idempotent operations (e.g., "transfer money"), replays can be catastrophic. The temporal aspect signifies that even if the content is valid, its timing renders it invalid.

### 2.4 Lack of Non-Repudiation

In regulated industries or systems requiring strict audit trails, the ability to definitively prove that a specific party sent a particular message and that the message was received unaltered is crucial.35 Standard JSON-RPC lacks inherent cryptographic signatures from the sender at the message layer. Consequently, a sender could potentially deny having sent a message, and a receiver would face difficulty in cryptographically proving its origin and integrity, hindering accountability and forensic analysis.35 Non-repudiation guarantees that the origin of data and its integrity can be proven, making it impossible for someone to claim they did not send or receive certain information.35 Non-repudiation typically involves a combination of digital signatures, audit logs, and time stamping.35 Digital signatures are considered a more powerful tool for publicly verifiable non-repudiation compared to Message Authentication Codes (MACs), which use symmetric keys.38

While HMAC provides authenticity (proof of origin to the *receiver*), true non-repudiation requires irrefutable proof to a *third party*.35 This distinction is critical for legal and audit contexts. The use of HMAC relies on a shared secret key. If a dispute arises, the sender could claim the key was compromised, or the receiver could have forged the message if they also possess the key. A digital signature, using asymmetric cryptography, provides a public-key verifiable proof that only the private key holder could have produced, making denial much harder. SJRM's use of HMAC contributes to strong authenticity, which is a component of non-repudiation, and is sufficient for many inter-service communication scenarios where trust is established between known parties. However, for external, legally binding transactions, a full digital signature might be preferred. The lack of non-repudiation poses significant challenges in regulated industries like FinTech and Healthcare, where verifiable audit trails and accountability are not just best practices but regulatory and operational requirements.42 This directly impacts trust and legal standing. Compliance frameworks (e.g., GDPR, HIPAA, PCI DSS) demand demonstrable data integrity and origin. Without non-repudiation, organizations face legal and financial penalties, and risk losing customer trust. SJRM's strong authenticity and integrity features, coupled with detailed logging, can significantly enhance auditability, even if full digital signatures are not used for every message.

### 2.5 Limitations of Transport-Layer Security (TLS/SSL)

While network-level security protocols such as TLS/SSL (Transport Layer Security / Secure Sockets Layer) are indispensable for encrypting the communication channel and providing endpoint authentication, their scope is limited to the transport layer.4 Once messages are decrypted at the application layer by an authorized service, or if the TLS tunnel is terminated at an API gateway, the individual JSON-RPC messages themselves remain vulnerable to the aforementioned application-layer attacks.4 Furthermore, in complex, multi-hop distributed systems, a single TLS tunnel between two services might not suffice to establish transitive trust or guarantee message integrity across multiple internal processing stages. An internal network segment compromise can expose unencrypted messages.1 Proxies and intermediaries can create problems for new HTTP extensions or alternative wire formats, potentially applying logic blindly or inferring malicious intent, leading to unreliable deployments.24 TLS primarily secures the channel; it does not prevent application-level attacks like SQL injection, cross-site scripting (XSS), or server-side request forgery (SSRF), which exploit vulnerabilities in how the application processes input.27

The termination of TLS at API gateways or load balancers, a common pattern in microservice architectures, creates a "cleartext zone" where messages are vulnerable before reaching their final destination. This necessitates a "defense in depth" strategy where security layers are applied at multiple levels, including the application message layer.4 API Gateways are crucial for managing external traffic, but they often terminate TLS to inspect, route, or transform requests. This means that within the internal network, messages might travel unencrypted or be re-encrypted with different keys. If an attacker compromises an internal service or network segment, the original TLS protection is lost. Message-level security ensures that even within this "cleartext zone," the integrity and authenticity of the message are maintained. The combination of TLS limitations and the sheer number of microservices with more interconnections and more communication links to be protected significantly amplifies the attack surface.36 This implies that a single point of failure in TLS termination or an internal network compromise can have widespread ripple effects across the entire distributed system. As systems become more distributed, the number of potential attack vectors increases exponentially. If each connection relies solely on TLS, and any single TLS termination point or internal network segment is compromised, the entire system becomes vulnerable to application-layer attacks. SJRM acts as a safety net, ensuring that even if the network layer is breached, the application layer remains protected. The imperative, therefore, is for a robust, per-message security wrapper that ensures the cryptographic integrity, authenticity, and freshness of every JSON-RPC interaction at the application level.

## 3. The SJRM Solution: Secure JSON-RPC Message Wrapper Protocol

The Secure JSON-RPC Message (SJRM) Wrapper Protocol is meticulously designed to augment standard JSON-RPC 2.0 by providing a robust, per-message security layer. SJRM achieves this by encapsulating conventional JSON-RPC request and response objects within an outer security envelope, which carries essential cryptographic metadata. This design allows for transparent integration with existing JSON-RPC implementations while introducing a critical layer of defense.

### 3.1 SJRM Message Structure

SJRM encapsulates the standard JSON-RPC message (whether a request, response, or notification) within a new, structured JSON object. This outer object introduces dedicated fields for security metadata, ensuring that the original JSON-RPC payload remains untouched and valid while gaining comprehensive protection.

The decision to keep the payload verbatim and unmodified is a deliberate design choice that ensures seamless integration with existing JSON-RPC parsers and logic. This minimizes the burden on developers to adapt their existing codebase, making adoption significantly easier.18 If SJRM required changes to the JSON-RPC payload itself, it would necessitate extensive refactoring of existing JSON-RPC clients and servers. By keeping the core payload untouched and adding security metadata as an outer envelope, SJRM acts as a transparent layer, allowing developers to add security without disrupting their core application logic. This directly supports the "seamless integration" benefit. Furthermore, the inclusion of

sjrm\_version is a forward-thinking design choice. It acknowledges that security protocols must evolve to counter new threats or incorporate stronger cryptographic primitives, ensuring the protocol's longevity and adaptability without requiring a "big bang" upgrade across all deployments.36 Security is not static; new vulnerabilities and cryptographic advancements emerge constantly. A versioning mechanism allows for phased rollouts of updates, enabling different services to operate with different versions of the SJRM protocol concurrently during a transition period. This is crucial for large, complex distributed systems where simultaneous updates are impractical.

**SJRM Message Structure**

|  |  |  |  |
| --- | --- | --- | --- |
| Field Name | Data Type | Description | Example Value |
| sjrm\_version | String or Integer | Explicit version identifier for the SJRM protocol itself, enabling controlled evolution and backward compatibility. | "1.0" or 1 |
| payload | Object | The verbatim JSON-RPC 2.0 object (request, response, or notification). It remains unmodified to preserve its original JSON-RPC semantics. | {"jsonrpc": "2.0", "method": "myMethod", "params": ["param1", "param2"], "id": 123} |
| security\_envelope | Object | A structured JSON object meticulously designed to carry cryptographic security elements. |  |
| security\_envelope.nonce | String | A cryptographically strong, pseudo-random string generated uniquely for each message. Fundamental in thwarting replay attacks by ensuring each message is processed only once. | "a\_unique\_random\_string\_xyz" |
| security\_envelope.timestamp | String or Number | The precise Coordinated Universal Time (UTC) timestamp when the message was generated, typically in ISO 8601 format or as Unix epoch milliseconds. Used to establish message "freshness." | "2025-07-31T10:30:00.000Z" or 1722297300000 |
| security\_envelope.hmac | String | The Hash-based Message Authentication Code, a cryptographic checksum providing message integrity and authenticity. A one-way hash of specific message components, keyed with a shared secret. | "calculated\_hmac\_value\_here" |
| security\_envelope.jwt | String | A JSON Web Token (JWT) used for robust sender authentication and to facilitate secure derivation or retrieval of the secret key required for HMAC verification. |  |

A clear, labeled block diagram illustrating a standard JSON-RPC message (e.g., {"jsonrpc": "2.0", "method": "subtract", "params": , "id": 1}) nested inside the SJRM wrapper would visually reinforce the new sjrm\_version and security\_envelope fields. Such a diagram is crucial for visual clarity, providing an immediate, intuitive understanding of how SJRM wraps the existing JSON-RPC message, demonstrating its non-invasive nature and the logical separation of concerns.

### 3.2 Core Security Mechanisms

SJRM meticulously integrates several industry-standard cryptographic techniques to provide its comprehensive security guarantees, building upon well-understood security principles:

#### 3.2.1 Message Integrity and Authenticity (HMAC)

**Mechanism:** SJRM leverages HMAC (Hash-based Message Authentication Code), typically implemented with strong cryptographic hash functions like HMAC-SHA256, to ensure both that the message content has not been altered and that the message genuinely originates from an authenticated sender possessing the correct secret key. HMACs are favored in high-performance distributed systems due to their efficiency compared to full digital signatures.45 Studies show that HMAC is the most efficient algorithm compared to RSA for signature generation and verification, especially at higher security levels, with very low time allocations.45

**Process:**

* **HMAC Input Construction:** The sender constructs a canonical representation of the data to be authenticated. This typically involves concatenating or otherwise deterministically combining the payload (the original JSON-RPC message), the nonce, and the timestamp fields from the SJRM message. It is vital that this input construction is identical on both sender and receiver sides.
* **HMAC Calculation:** Using a cryptographically strong shared secret key, the sender computes the HMAC over this constructed input. This secret key is known only to the sender and the intended receiver(s).
* **Inclusion:** The resulting HMAC value is then embedded within the hmac field of the security\_envelope.
* **Verification:** Upon receiving an SJRM message, the receiver independently reconstructs the exact same input data as the sender and re-calculates the HMAC using the same algorithm and the same shared secret key.
* **Validation:** The re-calculated HMAC is then rigorously compared to the hmac value provided in the incoming message. Any discrepancy, even a single bit, signifies that the message has been tampered with in transit or was not generated by an entity possessing the correct shared secret key, leading to immediate rejection.

The choice of HMAC over full digital signatures (which provide stronger non-repudiation) reflects a pragmatic trade-off for high-performance distributed systems.45 HMAC is computationally efficient, making it suitable for high-throughput microservices where latency is critical. This optimizes for the most common threats while maintaining performance. Digital signatures (asymmetric cryptography) are computationally more intensive than HMAC (symmetric cryptography). In a microservices environment with potentially millions of inter-service calls per second, even a slight increase in latency per call can lead to significant system-wide performance degradation. By choosing HMAC, SJRM prioritizes throughput and low latency, which are often paramount for internal microservice communication, while still providing strong integrity and authenticity within the trusted system boundary. The security of HMAC relies entirely on the secrecy and proper management of the shared symmetric keys.48 This highlights a critical dependency: even a perfectly designed protocol is only as secure as its weakest link, which in this case is the key management infrastructure. This points to the need for robust key management systems (KMS) and practices. If the shared secret key used for HMAC is compromised, an attacker can both forge messages and bypass integrity checks. This means that while HMAC itself is efficient, the overall security of SJRM is heavily reliant on external systems and processes for key generation, distribution, storage, and rotation. This is a crucial implementation consideration that must be emphasized.

#### 3.2.2 Replay Protection (Nonce and Timestamp)

**Mechanism:** SJRM employs a dual-layered approach for replay protection, combining a unique nonce with a precise timestamp. This robust combination addresses the challenges of asynchronous networks and guarantees message freshness.7

**Process:**

* **Nonce Generation & Uniqueness:** For every new SJRM message, the sender generates a cryptographically secure, pseudo-random nonce.7 The receiver maintains a stateful record (e.g., a distributed cache or database) of recently processed nonces for each sender. Any incoming message with a nonce that has already been seen and validated within a defined validity window is immediately rejected as a replay attempt. This necessitates efficient, possibly distributed, nonce management systems.49
* **Timestamp Validity:** The timestamp in the security\_envelope provides a temporal boundary for message validity. Receivers enforce a strict time window (e.g., rejecting messages older than N seconds or newer than M seconds in the future to account for clock skew). Messages falling outside this acceptable time window are discarded. This guards against both immediate replays and delayed replay attacks.8 The reliance on synchronized clocks is an important consideration.8
* **Combined Validation:** Both the nonce uniqueness and timestamp freshness checks are performed. A message must pass both criteria to be considered valid and processed. This layered approach significantly hardens the protocol against sophisticated replay attempts.

The combination of nonce and timestamp is crucial for distributed systems because of inherent network latency and clock drift.8 A nonce alone might be vulnerable if an attacker can predict it or if the system crashes and loses its nonce history. A timestamp alone might be vulnerable to clock skew or if an attacker can manipulate time. Their combination creates a stronger defense, covering both uniqueness and freshness. Distributed systems are asynchronous by nature. Messages can arrive out of order, and clocks on different machines can drift. A nonce ensures uniqueness, but if a system crashes and restarts, it might lose its "seen nonces" list, making it vulnerable to replays of messages sent before the crash. A timestamp provides freshness, but if clocks are not synchronized, a legitimate message might be rejected as "too old" or a replayed message might be accepted if the attacker's clock is skewed. The combination provides redundancy and robustness against these real-world distributed system challenges. The requirement for a stateful record of nonces and efficient, possibly distributed, nonce management systems 49 highlights a significant operational consideration. Implementing this efficiently at scale (e.g., using Redis Cluster) introduces complexity and potential performance bottlenecks that must be carefully designed and optimized. Every incoming message requires a nonce lookup. If this lookup is slow or becomes a bottleneck, the entire system's throughput will suffer. A distributed cache is essential to handle the volume and provide low-latency lookups, ensuring that the replay protection mechanism does not become a performance liability. The choice of distributed cache (e.g., Redis Cluster) introduces considerations around data consistency and replication models.52 While Redis offers eventual consistency by default, ensuring that all replicas are updated before acknowledging a write (e.g., using the

WAIT command) can improve consistency for nonces, which is crucial for preventing replay attacks across different service instances. If a nonce is processed by one instance and then another instance receives a replayed message before the nonce is propagated to its cache, the replay could be missed. Therefore, careful consideration of the cache's consistency model (e.g., strong vs. eventual consistency) and replication strategy is vital for the integrity of the replay protection.

#### 3.2.3 Sender Authentication and Secure Key Derivation (JWT)

**Mechanism:** JSON Web Tokens (JWTs) are utilized for securely authenticating the sender of the message and, critically, for facilitating the secure derivation or lookup of the shared secret key required for HMAC verification.11 JWTs are self-contained and digitally signed, making them ideal for stateless authentication in microservice environments.11

**Process:**

* **JWT Issuance:** A trusted Identity Provider (IdP) or an established secure key exchange mechanism issues a JWT to the sender service. This JWT is typically signed using the IdP's private key (for asymmetric signing) or a shared secret (for symmetric signing like HS256). The JWT's payload contains claims such as the sender's identity (sub claim), the issuer (iss), the audience (aud), and an expiration time (exp).
* **JWT Inclusion:** The sender includes this JWT directly within the jwt field of the security\_envelope of the SJRM message.
* **Receiver Validation:** Upon receiving the message, the receiver first validates the integrity and authenticity of the JWT itself by verifying its digital signature against the corresponding public key (or shared secret) of the IdP. This ensures the JWT hasn't been tampered with and was issued by a trusted entity.
* **Key Derivation/Lookup:** Once the JWT is validated, the receiver extracts relevant claims (e.g., the sub claim representing the sender's service ID). This information is then used to securely derive or retrieve the specific shared secret key (e.g., from a key management system or a pre-shared key store) that is necessary to verify the hmac of the current SJRM message. This dynamic key resolution, driven by authenticated identity, enhances security and simplifies key management in complex deployments.11

JWTs enable decoupled authentication where authentication data is stored separately and easily shared, and they are stateless.11 This is a significant advantage for scalability in microservices, as services do not need to maintain session state, reducing the risk of security incidents and simplifying session management. In traditional systems, session management often involves server-side state, which can be a bottleneck for scalability and a point of failure. JWTs, being self-contained and verifiable, allow services to authenticate requests without needing to query a central session store for every request. This statelessness is crucial for horizontally scaling microservices, as any instance of a service can process any request without prior session context. The reliance on JWTs necessitates adherence to JWT security best practices.12 This includes using strong signing algorithms (e.g., RS256 over HS256 where possible), proper expiration times, validating all claims (

iss, aud, exp, nbf, iat), and avoiding sensitive data in JWTs.11 Failure to do so can lead to vulnerabilities like replay attacks (if

jti and timestamps are not used), token invalidation issues, or information leakage. While JWTs offer many benefits, they are not a panacea. Their security is highly dependent on correct implementation. For instance, if JWTs are long-lived and there is no revocation mechanism, a compromised token remains valid until expiry. If claims are not properly validated, an attacker might forge a token with elevated privileges. This means that while SJRM leverages JWT, the underlying JWT implementation and its associated security practices are critical for the overall security posture.

### 3.3 Protocol Flow

The secure communication process using the SJRM protocol involves a well-defined sequence of operations on both the sender and receiver sides, ensuring that security checks are performed before the underlying JSON-RPC message is processed.

A detailed flowchart illustrating this process would depict the following sequence of operations:

**Sender Side:**

1. **Original JSON-RPC message:** The application generates a standard JSON-RPC request, response, or notification.
2. **Generate Cryptographically Secure Nonce & Current UTC Timestamp:** A unique, unpredictable nonce is generated, along with the precise UTC timestamp of message creation.
3. **Obtain/Derive Sender's JWT:** The sender service obtains a valid JWT from a trusted Identity Provider (IdP) or an internal key management mechanism.
4. **Serialize Payload, Nonce, Timestamp, JWT:** The original JSON-RPC payload, the generated nonce, the timestamp, and the JWT are combined into a canonical, deterministic representation. This step is crucial for consistent HMAC calculation.
5. **Calculate HMAC:** A Hash-based Message Authentication Code (HMAC) is computed over the serialized data using a cryptographically strong shared secret key.
6. **Construct SJRM Message:** The original JSON-RPC payload is encapsulated within the SJRM wrapper, along with the sjrm\_version and the security\_envelope containing the nonce, timestamp, HMAC, and JWT.
7. **Send SJRM Message:** The complete SJRM message is transmitted over the chosen transport layer (e.g., TLS/HTTP).

**Receiver Side:**

1. **Receive SJRM Message:** The SJRM message is received from the transport layer.
2. **Parse SJRM Message:** The received message is parsed to extract the payload, nonce, timestamp, hmac, and jwt from the security\_envelope.
3. **Validate JWT:**
   * The digital signature of the JWT is verified against the IdP's public key or shared secret to ensure its authenticity and integrity.
   * All relevant claims within the JWT (e.g., issuer (iss), audience (aud), expiration (exp), not-before (nbf), issued-at (iat)) are validated to confirm its validity and intended use.
   * **Decision Point:** If the JWT is invalid (e.g., bad signature, expired, incorrect claims), the message is immediately rejected.
4. **Determine/Derive Shared Secret Key:** If the JWT is valid, information extracted from its claims (e.g., the sender's service ID from the sub claim) is used to securely determine or derive the specific shared secret key required for HMAC verification. This key may be retrieved from a Key Management System (KMS) or a pre-shared key store.
5. **Check Nonce Uniqueness & Timestamp Freshness:**
   * The received nonce is checked against a record of recently processed nonces in a distributed nonce store to ensure it has not been used before by this sender within a defined validity window.
   * The timestamp is validated to ensure it falls within an acceptable time window (e.g., not too old or too far in the future) to account for clock skew.
   * **Decision Point:** If the nonce is not unique or the timestamp is outside the acceptable window, the message is rejected as a replay attempt.
6. **Re-calculate HMAC:** The receiver independently reconstructs the exact same input data (payload, nonce, timestamp) as the sender and re-calculates the HMAC using the same algorithm and the derived shared secret key.
7. **Compare Calculated HMAC with Received HMAC:** The re-calculated HMAC is rigorously compared to the hmac value provided in the received message.
8. **Decision Point:** If the HMACs do not match, the message is rejected as it indicates tampering or an unauthorized origin.
9. **Process Original JSON-RPC Payload:** If all security validations pass, the original JSON-RPC payload is deserialized and processed by the application logic.

The protocol flow demonstrates a sequential validation process (JWT first, then Nonce/Timestamp, then HMAC). This "early exit" strategy for invalid messages (e.g., rejecting on invalid JWT before nonce/HMAC checks) is a performance optimization, reducing unnecessary cryptographic computations for malicious or malformed messages.58 Cryptographic operations, especially signature verification and HMAC calculation, are computationally expensive. By performing cheaper checks (like JWT signature and basic claim validation) first, and then nonce/timestamp checks, the system can quickly discard invalid messages without incurring the full cost of all cryptographic operations. This improves overall system throughput and resilience against resource exhaustion attacks. The flow also implies a trust boundary around the Identity Provider (IdP) and the Key Management System (KMS). The security of the entire SJRM process fundamentally relies on the trustworthiness and availability of these external components. If the IdP is compromised, forged JWTs could be issued, leading to unauthorized key derivation and HMAC bypass. If the KMS is unavailable or compromised, legitimate services cannot retrieve keys, leading to service disruption or insecure communication. This highlights that SJRM, while strong, is part of a larger security ecosystem, and its effectiveness depends on the robustness of its dependencies.

### 3.4 Batch Request Handling

SJRM is designed with full compatibility for JSON-RPC 2.0 batch requests, an essential feature for optimizing network performance by sending multiple requests in a single transaction.18 When processing a batch, the entire array of JSON-RPC requests is treated as the singular payload within a single SJRM wrapper. The HMAC, nonce, and timestamp calculations thus cover the entire batch collectively. This ensures the integrity and authenticity of all encapsulated requests as a cohesive unit, preventing any individual request within the batch from being tampered with or replayed independently, while maintaining the efficiency benefits of batch processing.

Treating the entire batch as a single payload for security calculations (HMAC, nonce, timestamp) maintains the efficiency benefits of batch processing while providing atomic security guarantees. This prevents a scenario where an attacker might selectively tamper with or replay individual requests within a batch if they were secured separately. If each request in a batch required its own SJRM wrapper, the overhead would negate the performance benefits of batching. By wrapping the entire batch, SJRM ensures that the batch is treated as a single, atomic security unit. This means either the entire batch is valid and processed, or the entire batch is rejected, preventing partial processing of a compromised batch. This is crucial for maintaining transactional integrity in distributed systems. While efficient, this approach means that if any part of the batch is compromised (leading to HMAC failure), the entire batch is rejected. This might require more sophisticated error handling at the application layer to identify which specific request within the batch caused the failure, or to re-process valid requests from a rejected batch. A single invalid nonce or HMAC for a batch means all requests in that batch are discarded. This simplifies security validation but shifts the complexity of partial failures to the application. Developers might need to implement logic to re-send individual requests from a failed batch or to provide more granular error feedback to the client.

## 4. Key Benefits and Advantages of SJRM

The Secure JSON-RPC Message Wrapper Protocol offers a compelling suite of advantages for modern distributed applications, addressing critical security needs while maintaining operational efficiency:

### Comprehensive Message-Level Security

SJRM provides a robust, layered defense against the most prevalent application-layer attacks: message tampering, unauthorized access (impersonation), and replay attacks.27 This augments existing network-level security, offering granular protection directly where the data is processed, and addressing the limitations of relying solely on transport-layer security.4

### Guaranteed Data Integrity and Authenticity

By leveraging HMAC, SJRM cryptographically assures that every message's content remains unaltered from its origin and that it was genuinely sent by the asserted sender. This is foundational for trust in data exchange, especially in environments where data manipulation can have severe consequences.30

### Strong Replay Attack Mitigation

The intelligent combination of unique nonces and time-sensitive timestamps effectively neutralizes the threat of replay attacks 7, safeguarding against the re-execution of sensitive commands and preventing malicious duplication of operations. This is crucial for idempotent operations.

### Robust Sender Identity Verification

JWTs provide a stateless and verifiable mechanism for authenticating the origin of each message.11 This is essential for enforcing fine-grained authorization policies in dynamic microservice environments.44

### Lightweight and Optimized Performance

Despite its comprehensive security features, SJRM is designed to impose minimal computational and network overhead. The use of symmetric cryptography (HMAC) for integrity verification is computationally efficient, making it suitable for high-performance, high-throughput microservices where latency is a critical concern.13 Studies show that HMAC is the most efficient algorithm compared to RSA for signature generation and verification, especially at higher security levels, with very low time allocations.45 The overhead of cryptographic operations in RPC stacks can be a "predominant data center tax," making efficient solutions vital.15 The emphasis on lightweight and optimized performance is a strategic advantage. In a landscape where RPC communication suffers from significant overhead 13, a security solution that minimizes this overhead makes it economically and operationally viable for high-throughput microservices, preventing security from becoming a performance bottleneck or a "data center tax".15 If security measures introduce too much latency or consume excessive CPU, they can negate the performance benefits of microservices. This can lead to developers potentially bypassing security or scaling infrastructure unnecessarily. By ensuring minimal overhead, SJRM makes it easier for organizations to adopt and enforce security widely without compromising their core business objectives of speed and scalability.

### Seamless Integration and Adaptability

The wrapper-based design ensures that SJRM can be easily integrated into existing JSON-RPC 2.0 implementations without requiring extensive modifications to core application logic or a complete rewrite of services. It acts as an unobtrusive security layer.

### Future-Proof Design with Versioning

The explicit sjrm\_version field allows for planned evolution of the protocol, enabling new security features or adjustments to be introduced without forcing a simultaneous update across all communicating services, thus enhancing long-term maintainability.

### Enhanced Auditability and Non-Repudiation

The cryptographic evidence embedded within each SJRM message (HMAC, JWT) provides a verifiable audit trail, bolstering accountability and supporting non-repudiation claims for critical transactions and operations.35 While HMAC provides strong authenticity, full non-repudiation (irrefutable proof to a third party) typically requires asymmetric digital signatures.38 SJRM's mechanisms contribute significantly to auditability and non-repudiation within a defined trust boundary. The nuanced distinction regarding non-repudiation (HMAC vs. Digital Signatures) is crucial. While HMAC offers strong authenticity for inter-service communication, it is important to acknowledge that for legally binding external transactions, a full digital signature might be necessary. This suggests that SJRM is optimized for internal distributed system communication where shared secrets are managed securely, rather than public-facing, legally binding transactions requiring third-party verifiable proof. HMAC provides strong proof of origin to the intended recipient given a shared secret. For scenarios where a third party (e.g., a court) needs to verify the origin without relying on a shared secret, a digital signature (asymmetric cryptography) is typically required. SJRM's design is highly effective for securing internal microservice communication, where the trust model is often based on shared secrets and controlled environments.

## 5. Implementation Considerations

While the SJRM protocol provides a robust conceptual framework for message-level security, its effective production deployment necessitates careful consideration of several implementation best practices and operational challenges:

### 5.1 Robust Key Management

The security of HMAC relies entirely on the secrecy and proper management of the shared symmetric keys.48 Compromise of a single key could undermine the security of communications it protects. Secure generation, highly restricted storage (e.g., hardware security modules (HSMs), cloud key management services like AWS KMS, Azure Key Vault, Google Cloud KMS), secure distribution, and diligent rotation of these keys are paramount.48 NIST Special Publication 800-57 provides comprehensive guidance on cryptographic key management.48 JWTs facilitate dynamic key derivation or lookup, simplifying key management compared to static pre-shared keys for every service pair. This dynamic approach, driven by authenticated identity, enhances security and simplifies key management in complex deployments.11

The effectiveness of SJRM's cryptographic mechanisms (HMAC) is directly dependent on the robustness of external key management systems. This highlights a critical interdependency: a strong protocol cannot compensate for weak key management. If keys are hardcoded, poorly stored, or rarely rotated, they become a single point of failure. An attacker who gains access to a key can then forge messages or bypass integrity checks, rendering the HMAC protection useless. This means that organizations adopting SJRM must invest equally in a mature key management infrastructure. While JWT-driven key derivation simplifies distribution and lookup, it introduces complexity in managing the Identity Provider (IdP) and the Key Management System (KMS) themselves. This shifts the operational burden from managing individual shared secrets to managing a secure, highly available, and auditable IdP/KMS infrastructure. Dynamic key derivation is a significant improvement over static key distribution, especially in large microservice environments. However, it means the security of the entire system now hinges on the IdP and KMS. These systems become critical infrastructure that requires high availability, robust access controls, and regular auditing to prevent compromise.

### 5.2 Efficient Nonce Persistence and Validation

For truly effective replay protection, nonces must be stored and checked across all active instances of a receiving service.49 This often mandates the use of a highly available, distributed, and performant cache (e.g., Redis, memcached, or a specialized nonce store) with appropriate time-to-live (TTL) or expiry policies.52 The performance impact of this lookup must be considered for high-throughput systems, as continuous nonce validation can introduce latency.52 Distributed caching solutions like Redis provide horizontal scaling and fault tolerance by sharding data across multiple nodes and employing master-replica models with automatic failover, making them suitable for high-volume nonce management.52

The need for a highly available, distributed, and performant cache for nonces directly addresses a potential scalability bottleneck. If nonce checks were centralized or slow, they would become a choke point in high-throughput systems, negating SJRM's lightweight design.52 Every incoming message requires a nonce lookup. If this lookup is slow or becomes a bottleneck, the entire system's throughput will suffer. A distributed cache is essential to handle the volume and provide low-latency lookups, ensuring that the replay protection mechanism does not become a performance liability. The choice of distributed cache (e.g., Redis Cluster) introduces considerations around data consistency and replication models.52 While Redis offers eventual consistency by default, ensuring that all replicas are updated before acknowledging a write (e.g., using the

WAIT command) can improve consistency for nonces, which is crucial for preventing replay attacks across different service instances. If a nonce is processed by one instance and then another instance receives a replayed message before the nonce is propagated to its cache, the replay could be missed. Therefore, careful consideration of the cache's consistency model (e.g., strong vs. eventual consistency) and replication strategy is vital for the integrity of the replay protection.

### 5.3 Accurate Clock Synchronization

The efficacy of timestamp-based replay protection directly depends on accurate time synchronization between all communicating services.8 Significant clock skew can lead to legitimate messages being rejected or expired messages being accepted. Implementing Network Time Protocol (NTP) or similar time synchronization mechanisms across all system components is crucial.8 While NTP is widely used, it has limitations regarding accuracy (affected by network latency) and scalability (accuracy can decrease with more nodes).8 It is also susceptible to attacks like spoofing and replay attacks.8 Dynamic NTP algorithms are being developed to enhance precision and reliability in varying network conditions.57

The reliability of SJRM's replay protection (specifically timestamps) introduces a hidden dependency on external time synchronization services like NTP. This means the security of SJRM is indirectly tied to the security and accuracy of the underlying time infrastructure.8 If an attacker can manipulate the time reported by NTP, or if NTP itself is vulnerable to attacks, the timestamp validity checks in SJRM could be bypassed or lead to denial of service. This means that securing the time synchronization infrastructure is as important as securing the SJRM implementation itself. Achieving and maintaining accurate time synchronization across a large, geographically distributed system is a non-trivial operational burden.56 It requires continuous monitoring, adjustment, and potentially the use of advanced techniques like dynamic NTP to counteract clock drift and network latency. Clock drift is inherent in distributed systems. Even small discrepancies can lead to legitimate messages being rejected or replayed messages being accepted. This necessitates a dedicated operational focus on time synchronization, potentially involving specialized tools and expertise beyond the core application development.

### 5.4 Comprehensive Error Handling and Logging

Robust error handling for all security validation failures (e.g., invalid HMAC, expired timestamp, replayed nonce, invalid JWT) is essential. Detailed, actionable logging of these security events, integrated into a centralized logging and monitoring system, is vital for anomaly detection, incident response, and forensic analysis.28 NIST guidelines for microservices emphasize logging input validation errors, unexpected parameter errors, and requests for anomaly detection.36

Comprehensive logging of security validation failures transforms potential security incidents into observable events. This shifts the paradigm from reactive "breach response" to proactive "anomaly detection" and "threat hunting," enabling quicker incident response and minimizing damage.28 If security failures are silently dropped or poorly logged, an attacker could repeatedly try to exploit a vulnerability without being detected. Detailed logging provides the necessary data for security teams to identify patterns of attack, detect ongoing compromises, and respond effectively. Detailed logging, especially when combined with the cryptographic evidence in SJRM (HMAC, JWT), provides a verifiable audit trail that is critical for regulatory compliance and forensic analysis after a security incident. This moves beyond simple "logging" to "evidence collection." In the event of a breach or audit, logs are the primary source of truth. If the logs are comprehensive and contain cryptographic evidence (like the validated JWT claims or HMAC failure details), they provide irrefutable proof of what happened, who attempted what, and when. This is invaluable for meeting compliance requirements and for post-incident analysis.

### 5.5 Performance Benchmarking and Optimization

While SJRM is designed for efficiency, the real-world overhead introduced by cryptographic operations, nonce management, and JWT validation should be rigorously benchmarked in the specific deployment environment.58 Optimization strategies, such as batched nonce lookups or asynchronous key derivation, might be necessary for extremely high-throughput scenarios. HMAC is identified as the most efficient algorithm for signature generation and verification compared to RSA.45 Tools like Apache JMeter, Gatling, K6, and Artillery can be used for API protocol benchmarking, focusing on response time, throughput, error rates, and resource usage in controlled environments.58

The real-world overhead is highly dependent on the specific deployment environment.58 This implies that theoretical performance benchmarks (e.g., HMAC is faster than RSA) are a starting point, but actual performance will vary based on hardware, network conditions, specific cryptographic library implementations, and traffic patterns. A cryptographic algorithm's theoretical performance does not always translate directly to real-world application performance. Factors like CPU architecture, memory access patterns, network latency, and the efficiency of the underlying cryptographic library can all impact the final throughput. Therefore, rigorous benchmarking in the target environment is non-negotiable to ensure the solution meets performance requirements. Performance optimization is not a one-time task but an ongoing process. As microservices evolve and traffic patterns change, continuous benchmarking and optimization will be necessary to maintain the desired balance between security and performance. This aligns with DevSecOps principles.3 Microservice architectures are dynamic. New services are added, existing ones are updated, and traffic loads fluctuate. What is performant today might not be tomorrow. Integrating benchmarking into CI/CD pipelines ensures that performance regressions are caught early, and the system's security overhead remains within acceptable limits as it evolves.

## 6. Use Cases and Applications

The Secure JSON-RPC Message Wrapper Protocol is exceptionally well-suited for a broad spectrum of modern distributed system architectures and applications where the integrity, authenticity, and freshness of individual messages are paramount:

### Microservices Architectures

SJRM provides a consistent and verifiable security layer for inter-service communication within a cluster.36 This ensures that only authenticated and untampered messages are processed by internal services, even when operating within a supposedly trusted internal network segment, addressing the limitations of perimeter-only security.4 NIST guidelines for secure microservices emphasize the need for secure communication protocols and treating all microservices as non-trustworthy.36

### Multi-Agent Systems (MAS)

SJRM enables secure, verifiable, and trusted communication between autonomous software agents. This is particularly relevant in environments where agents may operate on different trust levels or across untrusted networks, ensuring message integrity and authenticity in dynamic, decentralized interactions.

### Internet of Things (IoT) Device Communication

SJRM secures command-and-control messages, sensor data streams, and firmware update requests originating from potentially exposed or resource-constrained IoT devices to backend processing systems. This is crucial where even minimal data manipulation can have significant physical consequences, providing integrity and authenticity for critical device-to-cloud communications.

### Financial Technology (FinTech) Transactions

SJRM ensures the cryptographic integrity and authenticity of critical financial transaction requests, payment instructions, and sensitive data exchanges.42 This addresses regulatory and operational requirements for non-repudiation and verifiable audit trails.35 Modernizing FinTech systems with microservices requires embedding security controls at the service level and integrating DevSecOps practices for compliance.42 For FinTech and Healthcare, SJRM is not merely a beneficial security feature; it is a critical enabler for regulatory and operational requirements.42 This implies that SJRM can directly contribute to an organization's ability to meet compliance standards (e.g., GDPR, HIPAA, PCI DSS) by providing auditable cryptographic proof of message integrity and origin. In highly regulated sectors, the cost of non-compliance (fines, reputational damage) is immense. SJRM's features directly address requirements for data integrity, authentication, and audit trails. By providing cryptographic evidence at the message level, it simplifies the process of demonstrating compliance during audits, making it a valuable investment beyond pure security.

### Healthcare and Personal Data Systems

SJRM protects highly sensitive patient data and ensures compliance with stringent data privacy regulations (e.g., HIPAA, GDPR) by securing data exchange between various healthcare applications and services. This minimizes risks of data breaches or unauthorized access, providing a verifiable layer of security for sensitive information.42

### API Gateways and Internal API Security

SJRM implements a consistent and enforceable security policy at the API gateway or directly at internal service endpoints. This adds a crucial layer of integrity and authentication before requests are routed to downstream services, complementing edge-level security and addressing the "unauthorized access" and "information leakage" threats.28

### Critical Infrastructure and Industrial Control Systems (ICS/OT)

In environments where the integrity and origin of every command are vital for operational safety and security, SJRM can provide an additional layer of assurance for communications within the control plane. This helps mitigate risks of data manipulation and unauthorized command injection. SJRM's applicability across multi-agent systems and IoT highlights its role in establishing trust in increasingly decentralized and heterogeneous environments. It provides a mechanism for entities with varying trust levels to communicate securely, fostering broader adoption of distributed paradigms in sensitive domains. As systems move beyond centralized control to distributed agents or vast IoT networks, the traditional trust boundaries blur. SJRM provides a per-message trust anchor, allowing individual messages to carry their own proof of origin and integrity. This is essential for building resilient and trustworthy systems where not all participants are equally trusted or operate within a single, secure perimeter.

## 7. Conclusion

The burgeoning landscape of distributed systems, driven by the agility and scalability of microservices, has concurrently unveiled a critical need for enhanced security mechanisms beyond traditional network-level protections.1 Standard JSON-RPC 2.0, while efficient and widely adopted, inherently lacks the per-message security necessary to counter contemporary threats such as tampering, impersonation, and replay attacks, particularly in complex, multi-component environments.

The Secure JSON-RPC Message (SJRM) Wrapper Protocol directly addresses this fundamental vulnerability. By introducing a transparent and cryptographically robust security envelope around each JSON-RPC message, SJRM systematically integrates essential security properties: message integrity and authenticity via HMAC, comprehensive replay protection through unique nonces and strict timestamps, and robust sender authentication leveraging JWT.7 This innovative approach ensures that every message processed by a distributed application is verifiably from an authenticated sender, has remained unaltered since its creation, and is fresh, thereby effectively mitigating a wide array of dangerous attack vectors.

SJRM’s design prioritizes lightweight overhead, ease of integration, and extensibility, making it a powerful and practical solution for developers and architects seeking to fortify the security posture of their JSON-RPC based distributed systems.13 The emphasis on SJRM being a practical solution due to its lightweight overhead, ease of integration, and extensibility highlights a crucial aspect of real-world security adoption. Security solutions, no matter how theoretically sound, will not be widely adopted if they are too complex, too expensive, or introduce unacceptable performance penalties. SJRM's design acknowledges this reality. Many security protocols are academically robust but practically difficult to implement or too resource-intensive for high-performance systems. SJRM's focus on being a "wrapper" and using efficient cryptographic primitives (HMAC) signals a design philosophy that balances strong security with the operational realities of modern distributed systems. This pragmatic approach is key to its potential for widespread adoption. As the complexity and interconnectedness of modern applications continue to grow, SJRM stands as a vital and pragmatic component in establishing trustworthy, resilient, and secure inter-service communication, safeguarding critical operations and sensitive data. The statement about the complexity and interconnectedness of modern applications continuing to grow implies an ever-evolving threat landscape. This positions SJRM not as a static solution, but as a foundational layer that can adapt (via versioning) to future threats, emphasizing the ongoing need for vigilance and adaptable security mechanisms in distributed systems. The security landscape is dynamic. New attack vectors and vulnerabilities constantly emerge. By building a protocol that is extensible (via

sjrm\_version) and focuses on fundamental cryptographic principles, SJRM provides a resilient foundation that can be updated and adapted over time without requiring a complete overhaul of the underlying architecture, thus future-proofing security investments.

## 8. References

* JSON-RPC 2.0 Specification: <https://www.jsonrpc.org/specification>
* RFC 2104: HMAC: Keyed-Hashing for Message Authentication: <https://datatracker.ietf.org/doc/html/rfc2104>
* RFC 7519: JSON Web Token (JWT): <https://datatracker.ietf.org/doc/html/rfc7519>
* SJRM Reference Implementation: <https://github.com/shalak97/Se>
* 1 GeeksforGeeks. (n.d.).  
  *Security in Distributed System*. Retrieved from <https://www.geeksforgeeks.org/ethical-hacking/security-in-distributed-system/>
* 27 F5. (2025, July 21).  
  *API Security Risks and Challenges*. Retrieved from <https://www.f5.com/company/blog/api-security-risks-and-challenges>
* 18 MoldStud. (2025, July 15).  
  *The Future of JSON-RPC in Microservices Architecture: Trends, Benefits, and Best Practices*. Retrieved from <https://moldstud.com/articles/p-the-future-of-json-rpc-in-microservices-architecture-trends-benefits-and-best-practices>
* 19 Wallarm. (2025, July 22).  
  *What is JSON-RPC*. Retrieved from <https://www.wallarm.com/what/what-is-json-rpc>
* 4 Stack Overflow. (n.d.).  
  *WCF: Transport vs. Message Security*. Retrieved from <https://stackoverflow.com/questions/5673283/wcf-transport-vs-message>
* 6 Research Trend. (n.d.).  
  *A Review Paper on Application Layer*. Retrieved from <https://www.researchtrend.net/ijeece/pdf/A-Review-Paper-on-Application-Layer-Anjali-3.pdf>
* 5 Saltzer, J. H., Reed, D. P., & Clark, D. D. (1984).  
  *End-to-End Arguments in System Design*. ACM Transactions on Computer Systems, 2(4), 277-288. Retrieved from <https://web.mit.edu/saltzer/www/publications/endtoend/endtoend.pdf>
* 28 ResearchGate. (2024, November).  
  *API Common Security Threats and Security Protection Strategies*. Retrieved from <https://www.researchgate.net/publication/386204013_API_Common_Security_Threats_and_Security_Protection_Strategies>
* 29 Oligo Security. (2025, January 6).  
  *Breaking Down OWASP Top 10 for Web Apps, Mobile, API, K8s, and LLMs*. Retrieved from <https://www.oligo.security/academy/breaking-down-owasp-top-10-for-web-apps-mobile-api-k8s-and-llms>
* 30 Wallarm. (2025, June 27).  
  *Parameter Tampering Attack*. Retrieved from <https://www.wallarm.com/what/parameter-tampering-attack>
* 31 Vectra AI. (n.d.).  
  *RPC Attacks*. Retrieved from <https://www.vectra.ai/modern-attack/attack-techniques/remote-procedure-call-rpc-attacks>
* 2 SciSpace. (2022, October).  
  *Security in Distributed System: A Review Perspective*. Retrieved from <https://scispace.com/pdf/security-in-distributed-system-a-review-perspective-3pfiehdr.pdf>
* 32 City University of Hong Kong. (n.d.).  
  *Security Analysis of Distributed Consensus Filtering Under Replay Attacks*. Retrieved from <https://scholars.cityu.edu.hk/files/183120931/168628466.pdf>
* 37 Naval Research Laboratory. (1994).  
  *A Taxonomy of Replay Attacks*. Retrieved from <https://apps.dtic.mil/sti/tr/pdf/ADA463948.pdf>
* 35 BitSight. (2025, April 3).  
  *Non-Repudiation in Cyber Security*. Retrieved from <https://www.bitsight.com/glossary/non-repudiation-cyber-security>
* 38 Wikipedia. (n.d.).  
  *Non-repudiation*. Retrieved from <https://en.wikipedia.org/wiki/Non-repudiation>
* 20 AWS. (n.d.).  
  *What is SSL/TLS Certificate?*. Retrieved from <https://aws.amazon.com/what-is/ssl-certificate/>
* 7 Wikipedia. (2025, July 14).  
  *Cryptographic Nonce*. Retrieved from <https://en.wikipedia.org/wiki/Cryptographic_nonce>
* 8 Number Analytics. (2025, June 14).  
  *Ultimate Guide: Clock Synchronization in Distributed Algorithms*. Retrieved from <https://www.numberanalytics.com/blog/ultimate-guide-clock-synchronization-distributed-algorithms>
* 60 Mills, D. L. (2012, May 14).  
  *NTP Security Analysis*. Retrieved from <https://www.eecis.udel.edu/~mills/security.html>
* 45 ResearchGate. (2025, April 26).  
  *Performance Comparison between Signature Cryptography: A Case Study on SNAP Indonesia*. Retrieved from <https://www.researchgate.net/publication/374377661_Performance_Comparison_between_Signature_Cryptography_A_Case_Study_on_SNAP_Indonesia>
* 46 NEC. (n.d.).  
  *Provably Secure MACs from Differentially-Uniform Permutations and AES-Based Implementations*. Retrieved from <https://www.nec.com/en/global/rd/tg/code/symenc/pcmacaes.html>
* 49 Kumar, A. (n.d.).  
  *Distributed Computing: Principles, Algorithms, and Systems*. Retrieved from <https://www.cs.uic.edu/~ajayk/Chapter16.pdf>
* 56 ViXra. (2022, April 1).  
  *Distributed Systems and Its Algorithms*. Retrieved from <https://vixra.org/pdf/2204.0094v1.pdf>
* 57 MDPI. (2024, January 22).  
  *A Dynamic Framework for Internet-Based Network Time Protocol (NTP) Accuracy*. Retrieved from <https://www.mdpi.com/1424-8220/24/2/691>
* 11 PMC. (2023, March 22).  
  *Enhancing Microservices Security with Token-Based Access Control Method Using JWT*. Retrieved from <https://pmc.ncbi.nlm.nih.gov/articles/PMC10052058/>
* 12 Aptori. (n.d.).  
  *JWT Security Best Practices Every Developer Should Know*. Retrieved from <https://www.aptori.com/blog/jwt-security-best-practices-every-developer-should-know>
* 47 IACR. (2025).  
  *Program: Crypto 2025*. Retrieved from <https://crypto.iacr.org/2025/program.php>
* 15 arXiv. (2025, May 16).  
  *Optimizing Intra-Container Communication with Memory Protection Keys for Microservices*. Retrieved from <https://arxiv.org/html/2505.07836v1>
* 3 The SAI. (n.d.).  
  *Security Gap in Microservices*. Retrieved from <https://thesai.org/Downloads/Volume15No12/Paper_18-Security_Gap_in_Microservices.pdf>
* 16 MDPI. (n.d.).  
  *Security and Privacy in Distributed Systems (Special Issue)*. Retrieved from <https://www.mdpi.com/journal/applsci/special_issues/51AI61FNCG>
* 36 NIST. (2020, May).  
  *NIST Special Publication 800-204A: Building Secure Microservices-based Applications Using Service-Mesh Architecture*. Retrieved from <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204A.pdf>
* 44 OWASP. (n.d.).  
  *Microservices Security Cheat Sheet*. Retrieved from <https://cheatsheetseries.owasp.org/cheatsheets/Microservices_Security_Cheat_Sheet.html>
* 61 DiVA Portal. (n.d.).  
  *Microservices: A Study of the Benefits and Challenges*. Retrieved from <https://www.diva-portal.org/smash/get/diva2:1528472/FULLTEXT01.pdf>
* 13 HotOS. (2025, May 14-16).  
  *Rethinking RPC Communication for Microservices-based Applications*. Retrieved from <https://ratul.org/papers/hotos2025-rpcs.pdf>
* 21 Internet Society. (n.d.).  
  *TLS Basics*. Retrieved from <https://www.internetsociety.org/deploy360/tls/basics/>
* 22 GeeksforGeeks. (2025, July 11).  
  *Difference Between Secure Socket Layer (SSL) and Transport Layer Security (TLS)*. Retrieved from <https://www.geeksforgeeks.org/computer-networks/difference-between-secure-socket-layer-ssl-and-transport-layer-security-tls/>
* 40 NIST. (2024, February 5).  
  *Digital Signature Standard (DSS) FIPS 186-4*. Retrieved from <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf>
* 41 ResearchGate. (n.d.).  
  *Enhancing the reliability of digital signatures as non-repudiation evidence under a holistic threat model*. Retrieved from <https://www.researchgate.net/publication/216628801_Enhancing_the_reliability_of_digital_signatures_as_non-repudiation_evidence_under_a_holistic_threat_model>
* 23 Encryption Consulting. (2025, May 9).  
  *Guide to TLS Encryption*. Retrieved from <https://www.encryptionconsulting.com/guide-to-tls-encryption/>
* 24 High Performance Browser Networking. (n.d.).  
  *Transport Layer Security (TLS)*. Retrieved from <https://hpbn.co/transport-layer-security-tls/>
* 25 NIST. (n.d.).  
  *SP 800-52 Rev. 1, Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations*. Retrieved from <https://csrc.nist.gov/pubs/sp/800/52/r1/final>
* 26 NIST. (n.d.).  
  *SP 800-52 Rev. 2, Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations*. Retrieved from <https://csrc.nist.gov/pubs/sp/800/52/r2/final>
* 62 Wikipedia. (2025, April 21).  
  *NIST SP 800-90A*. Retrieved from <https://en.wikipedia.org/wiki/NIST_SP_800-90A>
* 63 NIST. (2024, July 3).  
  *SP 800-90C Rev. 1, Recommendation for Random Bit Generator (RBG) Constructions*. Retrieved from <https://csrc.nist.gov/pubs/sp/800/90/c/4pd>
* 9 UEEx Technology. (2024, October 30).  
  *Cryptographic Nonce: What It Is and Its Purpose*. Retrieved from <https://blog.ueex.com/en-us/cryptographic-nonce/>
* 10 International Journal of Intelligent Systems and Applications in Engineering. (2024, February 2).  
  *An Adaptive Secure Timestamp-Based Replay Attack Detection System For Wsns*. Retrieved from <https://ijisae.org/index.php/IJISAE/article/view/4637>
* 55 GeeksforGeeks. (2025, July 23).  
  *Cross-Cutting Concerns in Distributed System*. Retrieved from <https://www.geeksforgeeks.org/system-design/cross-cutting-concerns-in-distributed-system/>
* 50 Wageningen University & Research. (n.d.).  
  *The most important effects both store layout and shelf design have on consumer behaviour*. Retrieved from <https://edepot.wur.nl/369091>
* 51 Investopedia. (n.d.).  
  *Blockchain*. Retrieved from <https://www.investopedia.com/terms/b/blockchain.asp>
* 52 Tinybird. (2025, June 7).  
  *Distributed Caching for Scalable Real-Time Systems*. Retrieved from <https://www.tinybird.co/blog-posts/distributed-caching-for-scalable-real-time-systems-sb>
* 58 DreamFactory. (2025, July 25).  
  *How to Benchmark API Protocols for Microservices*. Retrieved from <https://blog.dreamfactory.com/how-to-benchmark-api-protocols-for-microservices/>
* 59 Bermbach, D. (2020).  
  *Benchmarking the Performance of Microservice Applications: Challenges and Approach*. Retrieved from <https://dbermbach.github.io/publications/2020-acr-benchmarking.pdf>
* 14 DevOps School. (n.d.).  
  *A Complete Guide of RPC (Remote Procedure Call)*. Retrieved from <https://www.devopsschool.com/blog/a-complete-guide-of-rpc-remote-procedure-call/>
* 48 NIST. (2018, July 19).  
  *Recommendation for Key Management*. Retrieved from <https://csrc.nist.rip/projects/key-management/key-management-guidelines>
* 42 International Journal of Systems and Advanced Engineering. (2024, January-March).  
  *Security and Compliance Considerations in Modernizing FinTech Systems with Cloud-Native Microservices*. Retrieved from <https://www.ijsat.org/papers/2024/1/6070.pdf>
* 42 Citigroup. (n.d.).  
  *FinTech and the Criticality of Compliance*. Retrieved from <https://www.citigroup.com/global/insights/fintech-and-the-criticality-of-compliance>

#### Works cited

1. Security in Distributed System - GeeksforGeeks, accessed July 31, 2025, <https://www.geeksforgeeks.org/ethical-hacking/security-in-distributed-system/>
2. Security in Distributed System: A Review Perspective - SciSpace, accessed July 31, 2025, <https://scispace.com/pdf/security-in-distributed-system-a-review-perspective-3pfiehdr.pdf>
3. Security Gap in Microservices: A Systematic Literature Review, accessed July 31, 2025, <https://thesai.org/Downloads/Volume15No12/Paper_18-Security_Gap_in_Microservices.pdf>
4. WCF Transport vs Message - Stack Overflow, accessed July 31, 2025, <https://stackoverflow.com/questions/5673283/wcf-transport-vs-message>
5. END-TO-END ARGUMENTS IN SYSTEM DESIGN - MIT, accessed July 31, 2025, <https://web.mit.edu/saltzer/www/publications/endtoend/endtoend.pdf>
6. A Review Paper on Application Layer - Research Trend, accessed July 31, 2025, <https://www.researchtrend.net/ijeece/pdf/A-Review-Paper-on-Application-Layer-Anjali-3.pdf>
7. Cryptographic nonce - Wikipedia, accessed July 31, 2025, <https://en.wikipedia.org/wiki/Cryptographic_nonce>
8. Mastering Clock Sync in Distributed Systems - Number Analytics, accessed July 31, 2025, <https://www.numberanalytics.com/blog/ultimate-guide-clock-synchronization-distributed-algorithms>
9. Cryptographic Nonce: What It Is and Its Purpose - UEEx Technology, accessed July 31, 2025, <https://blog.ueex.com/en-us/cryptographic-nonce/>
10. An Adaptive Secure Timestamp-Based Replay Attack Detection ..., accessed July 31, 2025, <https://ijisae.org/index.php/IJISAE/article/view/4637>
11. Enhancing Microservices Security with Token-Based Access Control ..., accessed July 31, 2025, <https://pmc.ncbi.nlm.nih.gov/articles/PMC10052058/>
12. Advanced JWT Security Best Practices Every Developer Should Know - Aptori, accessed July 31, 2025, <https://www.aptori.com/blog/jwt-security-best-practices-every-developer-should-know>
13. Rethinking RPC Communication for Microservices ... - Ratul Mahajan, accessed July 31, 2025, <https://ratul.org/papers/hotos2025-rpcs.pdf>
14. A Complete Guide of RPC (Remote Procedure Call) - DevOpsSchool.com, accessed July 31, 2025, <https://www.devopsschool.com/blog/a-complete-guide-of-rpc-remote-procedure-call/>
15. RPCAcc: A High-Performance and Reconfigurable PCIe-attached RPC Accelerator - arXiv, accessed July 31, 2025, <https://arxiv.org/html/2411.07632v1>
16. Special Issue : Security and Privacy in Distributed Systems - MDPI, accessed July 31, 2025, <https://www.mdpi.com/journal/applsci/special_issues/51AI61FNCG>
17. Optimizing Intra-Container Communication with Memory Protection Keys: A Novel Approach to Secure and Efficient Microservice Interaction - arXiv, accessed July 31, 2025, <https://arxiv.org/html/2505.07836v1>
18. The Future of JSON-RPC in Microservices Architecture - MoldStud, accessed July 31, 2025, <https://moldstud.com/articles/p-the-future-of-json-rpc-in-microservices-architecture-trends-benefits-and-best-practices>
19. What is JSON-RPC? Meaning, Examples, Comparison - Wallarm, accessed July 31, 2025, <https://www.wallarm.com/what/what-is-json-rpc>
20. What Is An SSL Certificate? - SSL/TLS Certificate Explained - AWS, accessed July 31, 2025, <https://aws.amazon.com/what-is/ssl-certificate/>
21. What is TLS & How Does it Work? - Internet Society, accessed July 31, 2025, <https://www.internetsociety.org/deploy360/tls/basics/>
22. Difference Between Secure Socket Layer (SSL) and Transport Layer ..., accessed July 31, 2025, <https://www.geeksforgeeks.org/computer-networks/difference-between-secure-socket-layer-ssl-and-transport-layer-security-tls/>
23. A Comprehensive Guide to TLS Encryption | Encryption Consulting, accessed July 31, 2025, <https://www.encryptionconsulting.com/guide-to-tls-encryption/>
24. Networking 101: Transport Layer Security (TLS) - High Performance ..., accessed July 31, 2025, <https://hpbn.co/transport-layer-security-tls/>
25. SP 800-52 Rev. 1, Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations | CSRC, accessed July 31, 2025, <https://csrc.nist.gov/pubs/sp/800/52/r1/final>
26. SP 800-52 Rev. 2, Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations | CSRC, accessed July 31, 2025, <https://csrc.nist.gov/pubs/sp/800/52/r2/final>
27. API Security Risks and Challenges | F5 - F5 Networks, accessed July 31, 2025, <https://www.f5.com/company/blog/api-security-risks-and-challenges>
28. (PDF) API Common Security Threats and Security Protection ..., accessed July 31, 2025, <https://www.researchgate.net/publication/386204013_API_Common_Security_Threats_and_Security_Protection_Strategies>
29. Breaking Down OWASP Top 10 for Web Apps, Mobile, API, K8s ..., accessed July 31, 2025, <https://www.oligo.security/academy/breaking-down-owasp-top-10-for-web-apps-mobile-api-k8s-and-llms>
30. Parameter Tampering Attack - What is it? - Wallarm, accessed July 31, 2025, <https://www.wallarm.com/what/parameter-tampering-attack>
31. What is an RPC Attack + How to Detect It - Vectra AI, accessed July 31, 2025, <https://www.vectra.ai/modern-attack/attack-techniques/remote-procedure-call-rpc-attacks>
32. API vulnerability, unprotected devices - Security Boulevard, accessed July 31, 2025, <https://securityboulevard.com/2025/07/api-vulnerability-unprotected-devices/>
33. Security Analysis of Distributed Consensus Filtering ... - CityU Scholars, accessed July 31, 2025, <https://scholars.cityu.edu.hk/files/183120931/168628466.pdf>
34. End-to-end principle - Wikipedia, accessed July 31, 2025, <https://en.wikipedia.org/wiki/End-to-end_principle>
35. What is Non-repudiation in Cyber Security? | Bitsight, accessed July 31, 2025, <https://www.bitsight.com/glossary/non-repudiation-cyber-security>
36. Building Secure Microservices-based Applications Using Service ..., accessed July 31, 2025, <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204A.pdf>
37. A Taxonomy of Replay Attacks - DTIC, accessed July 31, 2025, <https://apps.dtic.mil/sti/tr/pdf/ADA463948.pdf>
38. Non-repudiation - Wikipedia, accessed July 31, 2025, <https://en.wikipedia.org/wiki/Non-repudiation>
39. Evidence and Non-repudiation - CiteSeerX, accessed July 31, 2025, <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=40dc9d6cb2207a71ea39ef2e0cfab668688146a3>
40. Digital Signature Standard (DSS) - NIST Technical Series Publications, accessed July 31, 2025, <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf>
41. (PDF) Enhancing the reliability of digital signatures as non ..., accessed July 31, 2025, <https://www.researchgate.net/publication/216628801_Enhancing_the_reliability_of_digital_signatures_as_non-repudiation_evidence_under_a_holistic_threat_model>
42. Modernizing Legacy FinTech Systems through Cloud-Native ... - IJSAT, accessed July 31, 2025, <https://www.ijsat.org/papers/2024/1/6070.pdf>
43. Fintech and the criticality of compliance - Citi, accessed July 31, 2025, <https://www.citigroup.com/global/insights/fintech-and-the-criticality-of-compliance>
44. Microservices Security - OWASP Cheat Sheet Series, accessed July 31, 2025, <https://cheatsheetseries.owasp.org/cheatsheets/Microservices_Security_Cheat_Sheet.html>
45. (PDF) Performance Comparison between Signature Cryptography ..., accessed July 31, 2025, <https://www.researchgate.net/publication/374377661_Performance_Comparison_between_Signature_Cryptography_A_Case_Study_on_SNAP_Indonesia>
46. PC-MAC-AES: Symmetric Cryptography Research | NEC, accessed July 31, 2025, <https://www.nec.com/en/global/rd/tg/code/symenc/pcmacaes.html>
47. Crypto 2025 Program - IACR, accessed July 31, 2025, <https://crypto.iacr.org/2025/program.php>
48. Key Management Guidelines - Key Management | CSRC, accessed July 31, 2025, <https://csrc.nist.rip/projects/key-management/key-management-guidelines>
49. Chapter 16: Authentication in Distributed System, accessed July 31, 2025, <https://www.cs.uic.edu/~ajayk/Chapter16.pdf>
50. The effects of in-store layout- and shelf designs on consumer behaviour - WUR eDepot, accessed July 31, 2025, <https://edepot.wur.nl/369091>
51. Blockchain Facts: What Is It, How It Works, and How It Can Be Used - Investopedia, accessed July 31, 2025, <https://www.investopedia.com/terms/b/blockchain.asp>
52. Distributed Caching for Scalable Real-Time Systems - Tinybird, accessed July 31, 2025, <https://www.tinybird.co/blog-posts/distributed-caching-for-scalable-real-time-systems-sb>
53. Distributed Caching - Redis, accessed July 31, 2025, <https://redis.io/glossary/distributed-caching/>
54. Scale with Redis Cluster | Docs, accessed July 31, 2025, <https://redis.io/docs/latest/operate/oss_and_stack/management/scaling/>
55. Cross-Cutting Concerns in Distributed System - GeeksforGeeks, accessed July 31, 2025, <https://www.geeksforgeeks.org/system-design/cross-cutting-concerns-in-distributed-system/>
56. Clock Synchronization in Distributed Systems - viXra.org, accessed July 31, 2025, <https://vixra.org/pdf/2204.0094v1.pdf>
57. A Dynamic Framework for Internet-Based Network Time Protocol, accessed July 31, 2025, <https://www.mdpi.com/1424-8220/24/2/691>
58. How to Benchmark API Protocols for Microservices, accessed July 31, 2025, <https://blog.dreamfactory.com/how-to-benchmark-api-protocols-for-microservices>
59. Benchmarking the Performance of Microservice ... - David Bermbach, accessed July 31, 2025, <https://dbermbach.github.io/publications/2020-acr-benchmarking.pdf>
60. NTP Security Analysis, accessed July 31, 2025, <https://www.eecis.udel.edu/~mills/security.html>
61. unraveling microservices - DiVA portal, accessed July 31, 2025, <https://www.diva-portal.org/smash/get/diva2:1528472/FULLTEXT01.pdf>
62. NIST SP 800-90A - Wikipedia, accessed July 31, 2025, <https://en.wikipedia.org/wiki/NIST_SP_800-90A>
63. SP 800-90C, Recommendation for Random Bit Generator (RBG ..., accessed July 31, 2025, <https://csrc.nist.gov/pubs/sp/800/90/c/4pd>