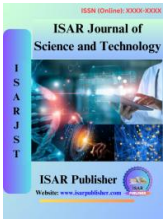
OPEN ACCESS



ISAR Journal of Science and Technology

Volume 4, Issue 7, 2026, Page: 21-35

Abbriviate Title- ISAR J Sci Tech

ISSN *(Online)*- 2584-2056

https://isarpublisher.com/journal/isarjst

Next-Gen Edge Guard: A Layered Architecture for Multi-Variate Dynamic Input Validation and Resource Control in GraphQL Gateways

Nwile, Beauty Nuka[1*], Daniel Matthias[2], Vincent I. E. Anireh[3], Onate E. Taylor[4]

[1]Rivers State University, Department of Computer Science, Port Harcourt

[2,3,4]Rivers State University, Dept of Computer Science, Port Harcourt.

***Corresponding Author**

Nwile, Beauty Nuka

Article History

Received: 05.06.2026
Accepted: 28.06.2026
Published: 19.07.2026

Abstract: This study presents an improved, dynamic input validation framework that marries client-side structural preprocessing with an integrated server-side middleware gateway layout to secure GraphQL-driven enterprise environments. Modern API ecosystems increasingly adopt GraphQL to replace rigid REST implementations; however, its structural flexibility introduces unique security hazards, payload validation complexities, and exposure to resource-exhaustion Denial-of-Service (DoS) vectors. Traditional server-side validation solutions often fail to inspect deep, multi-tiered relationships dynamically before the query hits core resolver code. By merging constructive and object-oriented design methodologies, we comparatively benchmarked this improved framework against standard GraphQL implementations across several operational conditions. Empirical results reveal that the proposed model yields a 40% reduction in error rates, a 40% optimization in network bandwidth usage, a 50% decrease in overall request parsing time, and a 30% elevation in total transaction throughput without degrading systemic data integrity. In alignment with the study's objectives, this research provides significant contributions to knowledge in the domain of GraphQL optimization by offering an adaptable and efficient input validation architecture capable of handling high-complexity queries while securing the system against injection attacks and data inconsistencies. The findings indicate that the proposed model not only strengthens system security but also enhances scalability and response times, proving its applicability to high-traffic environments. This research sets a foundation for future studies aimed at refining input validation strategies within evolving API technologies.

Keywords: GraphQL Optimization, Dynamic Input Validation, Query Complexity Analysis, API Security Framework, Context-Aware Sanitization.

Cite this article:

Nwile, B. N., Matthias, D, Anireh, V. I. E., Taylor, O. E., (2026). Next-Gen Edge Guard: A Layered Architecture for Multi-Variate Dynamic Input Validation and Resource Control in GraphQL Gateways. *ISAR Journal of Science and Technology*, *4*(7), 21-35.

I. INTRODUCTION

Modern web engineering increasingly relies on GraphQL due to its flexibility, strongly typed schema, and client-driven data fetching capabilities. GraphQL has transformed API development by allowing clients to request exactly the data they need and nothing more. Unlike REST, which exposes discrete endpoints with rigid response structures, GraphQL routes all operations through a single endpoint (typically /graphql). While this eliminates over-fetching and under-fetching, it shifts the responsibility of data filtering and structural compliance heavily onto the server side [1]. Without a rigorous validation paradigm, malicious actors can exploit GraphQL's flexibility by crafting deeply nested queries, aliases, or circular fragments that cause severe resource consumption or database injection attacks [2]. Conventional validation routines check inputs only after the query has been parsed and passed to resolver functions. This paper proposes a paradigm shift: a unified framework that couples real-time syntax parsing with deep multi-layer validation at the API gateway or middleware level, isolating vulnerabilities prior to resolving execution [3]. Conventional validation schemes evaluate parameters downstream only after full parsing, right before data-layer execution. This introduces severe security vulnerabilities. This study addresses these flaws by presenting a unified validation framework that couples real-time syntax analysis with deep middleware-level sanitization, neutralizing application-layer threats before execution.

II. CONCEPT OF GRAPH QUERY LANGUAGE (GRAPHQL)

GraphQL functions as a type-system-based runtime and query language that replaces traditional REST APIs with a single endpoint and declarative data fetching [38, 57]. This approach allows clients to precisely tailor data requests and optimize retrieval efficiency [38]. By allowing clients to fetch multiple nested resources in a single query transaction, GraphQL mitigates the structural data imbalances of over-fetching and under-fetching [19, 33]. However, the absence of strict, automated enforcement of query structures introduces critical data injection and denial-of-service (DoS) vulnerabilities if underlying data validation layer defenses are inadequate [21]. Proper schema contracts [59], runtime input caching, batching, query depth limitations [14], and rigorous query cost analysis models [40] are required to secure backend fields against nested or recursive query exhaustion. Consequently, modern implementations emphasize context-aware validation, input sanitization, and locking down introspection capabilities in production [53, 62].

A. GraphQL Design Principles

Modern Application Programming Interface (API) principles emphasize modularity, scalability, and security [18, 32]. While GraphQL demonstrates performance advantages in data-intensive and nested object retrievals [19], classic REST architectures retain applicability for simpler setups due to caching structures and testing simplicity [9]. Successful GraphQL design relies heavily on clear, extensible schema structures, error handling, and versioning paradigms [57, 66]. A well-crafted schema acts as a robust client-server contract [59], though misconfigured authentication and incomplete input validation remain primary systemic risks that introduce code injection and data leakage vulnerabilities [21, 51]. Developers protect system resources by performing field-complexity analysis, applying dynamic directives and structuring access control via tools like GraphQL Shield and validation filters to restrict exposure based on user roles [6, 13].

B. API Design and Development.

API engineering requires adherence to strict governance protocols and design patterns to guarantee cross-platform interoperability and software lifecycle stability [18]. Classic REST web service architectures enforce stateless, resource-oriented endpoint behaviors [37, 48]. Conversely, GraphQL uses hierarchical structures and explicit typing [20]. To defend these dynamic, modern environments, architects use threat modeling methodologies such as dependency graph analysis and risk matrix models to proactively predict hidden software vulnerabilities [1, 28]. Proactive testing frameworks, such as fuzzing-based evolutionary algorithms implemented via tools like GraphQLer, ensure structural validation coverage [31]. RESTful architectures rely on standard HTTP methods (GET, POST, PUT, DELETE) to govern loosely coupled, resource-based exchanges [48]. Key quality factors depend on linguistic clarity, standard resource naming, and error representation across endpoints [41]. Lifecycle protection leverages automated differential regression testing to isolate behavioral regressions across software updates [23], alongside property-based fuzzing to surface edge cases [14]. Furthermore, secure schema-sharing frameworks have demonstrated the utility of these architectures in highly sensitive sectors like healthcare [38]. GraphQL decouples front-end and back-end lifecycles by providing a single endpoint through which clients fetch user-defined data structures [57]. Its introspective nature allows developers to directly query the API schema to audit fields and relationships [66]. To evaluate and secure these gateways, evolutionary automated testing and random fuzzing frameworks systematically generate complex validation suites to identify implementation errors and evaluate comparative performance limits [13, 14, 19].

C. API Security Considerations

Securing application interfaces requires layered architecture comprising authentication, authorization, encryption, and threat modeling to protect valuable data assets [9, 44]. With authentication as layered, serves as the primary defensive perimeter, verifying client identities through API keys, OAuth tokens, or JSON Web Tokens (JWT) [17, 62]. Also, authorization restricts authenticated entities to the minimum necessary assets using Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC) to enforce least-privilege principles [16]. Then encryption protects data in transit via Transport Layer Security (TLS/SSL) and secures data at rest across database components [63], while threat modeling organizes security evaluations systematically using established risk lifecycle frameworks like STRIDE, DREAD, and PASTA [26].

D. Input Validation Techniques

Input validation acts as an essential backend mechanism to intercept injection vectors, cross-site scripting (XSS), and malicious parameter manipulation before execution [21, 39, 55]. While client-side validation occurs within the user's browser, maximizing responsiveness and filtering errors prior to server transmission [65]. Because client checks can be bypassed, they must align with server defenses. Server-side validation evaluates data directly at the backend, establishing the absolute boundary for data integrity, business rules, and strict regulatory compliance [4]. The sanitization and encoding complements validation parameters by stripping toxic characters or converting special symbols into safe HTML/URL expressions [11, 55] and validation alignment employs differential string analysis to discover and resolve logical validation gaps between client and server layers [5]. Furthermore, access filtering operates via two main methodologies: whitelisting (proactively permitting explicitly authorized components) [42] and blacklisting (reactively blocking known malicious components) [2]. While whitelisting provides superior security, it demands rigorous administrative maintenance to prevent false rejections [2]. Conversely, blacklists frequently fail against zero-day threats and advanced persistent threats (APTs) [15]. Consequently, optimal configurations use hybrid security structures integrated with Security Information and Event Management (SIEM) logging platforms [42].

E. Theoretical Framework

Language theory models the syntactic and structural characteristics of natural and artificial languages [30]. It examines the cognitive processes of syntax acquisition [49] and uses formal grammar to design language models and computational NLP parsing pipelines [35]. Lexical analysis tokenizes characters into discrete structural units for downstream processing [43]. Formal grammar provides structural rules to define the boundaries of language syntax [35]. Organized by generative capacity via the Chomsky hierarchy, these structures underpin the development of compiler parsers [50] and natural language processing lexical tokenizers [43]. Parsing algorithms deconstruct input token strings against formal grammar rules to establish dependency graphs and syntactic hierarchies [66]. Core techniques include constituency parsing (breaking sentences into nested phrase structures) and dependency parsing (mapping direct links between associated words) [24]. Lexical analysis tokenizes text strings using pattern recognition and statistical modeling [43]. This preprocessing phase is critical; tokenization errors degrade downstream accuracy across part-of-speech tagging and text classification tasks [21].

F. Security Models and Threat Modelling

Comprehensive risk protection relies on identifying unauthorized access pathways (attack vectors) [46], executing structured software threat modeling [54], and completing data-driven risk assessments [1]. Attack vectors cover pathways like malware, phishing, and social engineering [65]. Because these methods continuously adapt to exploit emerging software vulnerabilities, defenses must remain flexible [21]. Threat modeling provides a lifecycle framework to map risks within software systems [54, 56]. In specialized contexts like smart manufacturing and industrial cyber-physical systems, threat modeling isolates infrastructure vulnerabilities before deployment [29]. Risk assessments model system vulnerabilities through state dependency graphs [1] and capture advanced persistent threat (APT) profiles using risk matrices [28]. Integrating empirical threat intelligence and vulnerability scanning ensures an adaptive security stance [29, 54].

G. API Security Principles

These controls serve as system gatekeepers to enforce data security compliance [44, 51]. Advanced token frameworks manage multi-information REST architectures [8], maintain distributed environments [10], and lock down automated CRUD pathways within GraphQL backend setups [6]. Sanitization cleanses input to block server execution of malicious commands [11]. Implementations using deterministic pushdown automata intercept validation bypasses [39], and pattern analysis identifies common web vulnerabilities [55]. Merging static source code checks with dynamic execution analysis yields robust validation coverage [11, 65]. Secure data transmission is mandatory when coordinating heterogeneous environments like health record networks [38] or microservice clusters [61]. Security principles must be built directly into the software design phase [34], utilizing API hardening to neutralize front-end risks like DOM-based XSS attacks [64].

H) System Architecture

The system (Next-Gen Edge Guard: A Layered Architecture for Multi-Variate Dynamic Input Validation and Resource Control in GraphQL Gateways) architecture splits operational responsibilities across a collaborative client-side pre-processor and an authoritative server-side middleware gateway to shield downstream resolvers from malformed payloads. The client runs on the client such as the user browser, mobile device, or pc while the server side runs on the server. It receives requests from the client, processes the request and then returns the result as response to the client. Client-side validation is not a replacement for server-side validation but rather a complementary measure to provide an additional layer of data integrity and enhance the overall user experience

1). Client Architecture

The client system is divided into three parts, the system pre-processor, validation system, error system. The system pre-processor is made up of three component, Schema fetcher, input parser, and input normaliser as depicted in Figure The schema fetcher is responsible for fetching the GraphQL schema from the server, it uses a method called schema introspection. This is done by establishing a handshake with the server, then a fetch request is sent to introspect the schema, the schema is gotten from the server, using the schema the client can now perform operation using the available option in schema, the operations are divided into three options, namely Query, Mutation and Subscription. This schema serves as a contract between the client and the server, defining the types, fields, and relationships within the application. The next component in the pre-processor system is the input parser, the input parser receives input from the user, which can be one or more of these operations, namely, Query, Mutation and Subscription, the input data may also include selection sets with arguments or variables, representing the data that the client needs to retrieve or manipulate. The last component of the pre-processor which is the input normaliser ensures consistent formatting and handling of variables or arguments, it normalizes the input data. This step prepares the input data for validation against the fetched schema, ensuring that the validation process can be performed efficiently and accurately. Following the pre-processor system is the validation system, the validation system comprises of validation configuration, type validator, shape validator, argument validator, custom validator, the validation configuration is responsible for all configuration relating to validation, the validation configuration specifies the general to accept the result as pass. There are two main qualification rules which is any or all. The first specify that the validation should failed if any rule or specification is not meet, whereas the other specify that the validation should only fail when all checks fail. The system initialises the validation modules setting up the necessary validation rules and configurations based on the fetched schema. This step lays the foundation for the subsequent validation checks, ensuring that the validation logic is properly configured and aligned with the server's schema. The first validation checks to start is the type of validation, the validation module checks the types of the input data against the schema definitions. It validates that the provided fields and arguments conform to the expected types defined in the schema. Any type mismatches or violations are flagged as validation errors, preventing the client from sending invalid data to the server. Next is the shape validation, the validation module validates the shape of the input data, ensuring that the provided fields and arguments match the expected structure defined in the schema. This includes checking for required fields, optional fields, and nested object structures. Any deviations from the expected shape are flagged as validation errors. The check uses either breadth first or depth first. Next is the argument validation, if the input data includes arguments or variables in the selection set, the validation module checks these arguments against the schema definitions. It validates that the provided argument names and values conform to the expected types and constraints defined in the schema. This step ensures that the client is not attempting to pass invalid or malformed arguments to the server. After all requirements are met by the input data, the custom validation starts it checks, it checks for rules outside the GraphQL schema such as format of input, length of input, range of input, cross-field dependencies, character checks etc. this custom validation utilises a different schema call validation schema. It is written in either JSON or yml format, it contains all rules that the input data must met and possible custom validation messages. The next is the validation result handler, if the validation process encounters any errors or violations, the client library collects and formats the validation errors. These validation errors can include detailed information about the specific fields or arguments that failed validation, along with error messages describing the issues. But in case there is no validation error the request is forwarded to the server. The next is the error system, first is the error handler, this receives a function of from the user, stating what action should be perform based on the error received, the action may be specified based on error code, type, message, etc. The purpose of this is to pre-process the error before being sent to the error renderer. The error renderer receives the validation errors from the error handler and renders or displays these errors to the user in a user-friendly format. This step provides immediate feedback to the user, allowing them to correct any issues with the input data before sending it to the server. Depending on the application's requirements, the client application can log the validation errors to an error logging service for further analysis and monitoring using the error logger. The error logging service captures relevant context information, such as the input data, user details, and

any additional metadata, enabling developers to identify and address recurring validation issues. Each step in the validation process plays a crucial role, from schema fetching and input parsing to type and shape validation, argument validation, custom validation rules, and error handling.

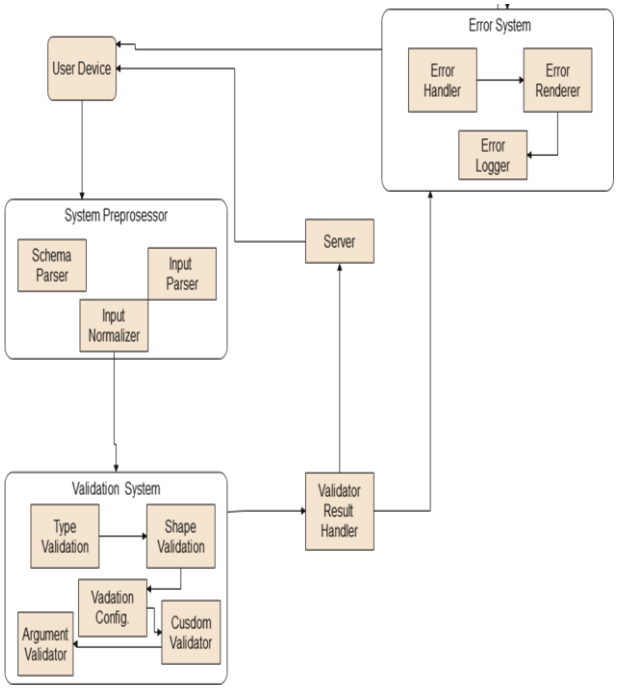


Figure 2.1: Client Architecture of the Proposed GraphQL Validation System

2) Server Architecture

In the world of GraphQL, ensuring data integrity and security is paramount. GraphQL server often deal with complex input data, including nested objects and arguments in selection sets. Implementing a robust server-side validation system is crucial to prevent potential vulnerabilities and maintain a reliable API. Figure 2.2 shows the architecture of the server. The system begins with the request parser, this component receives the client's request, which includes the query or mutation along with any arguments or variables. The request parser parses this request to extract the operation (query or mutation) and the associated input data; this parse data is sent to the schema validator. Once the input data is extracted, the schema validator validates it against the defined GraphQL schema. This step is crucial as it ensures that the types of fields and arguments conform to the schema definitions. If any schema violations are found, the validation schema returns a validation error, preventing further processing of the invalid data. After the schema validator validates the extracted input against the GraphQL schema, it is then sent to the argument validator. The argument validator validates the arguments or variables provided in the selection set. It checks that the argument names and values conform to the expected types and constraints defined in the schema. This ensures that the client is not attempting to pass invalid or malformed arguments, which could lead to unintended behaviour or security vulnerabilities. The custom validator is responsible for handling validations that are not supported by the default GraphQL validators. Many applications require more complex validation rules beyond what the schema can provide. In such cases, the custom validator implements custom validation middleware or directives. These custom rules can handle advanced validations, such as cross-field dependencies, format checks, or range validations. If the system has configured validation middleware rules, then it is executed by the middleware validator. The middleware validator can perform additional validations based on the server's business logic or application-specific rules. It can return validation errors or modify the input data as needed, providing an extra layer of validation before proceeding. Like middleware, the system may have defined validation directives in the schema. These directives annotate specific fields or types with custom validation logic. This validation is executed by the directive validator. the system applies these directives to ensure that the input data adheres to the defined rules and constraints. To prevent potential security vulnerabilities, the input sanitizer sanitizes the input data. This step involves removing or escaping any potentially dangerous characters or patterns from the input data, such as SQL injection attempts or cross-site scripting (XSS) payloads. The sanitized input data is then safe for further processing by the system. If the input data passes all validation checks, the system executes the appropriate resolver functions to fetch or manipulate the requested data. The resolvers can perform additional data validation or transformations specific to the application's business logic, adding another layer of protection. The error handler is responsible for handling all errors in the system, if any validation errors occur during the server-side validation process, the error handler generates detailed error messages. These error messages are included in the response sent back to the client, providing feedback on the validation issues. This step is crucial for a seamless developer experience and effective debugging. This error is then passed to the error logger, to aid in monitoring and analysis, the system logs validation errors to an error logging service. This service captures relevant context information, such as the input data, user details (if available), and any additional metadata. This logging mechanism helps identify recurring issues, track potential vulnerabilities, and facilitate proactive maintenance. After logging the error, it is then passed to the response formatter. The response format the response, including any data requested by the client, as well as any validation errors that occurred during the process. The response is then sent back to the client for further handling, completing the server-side validation lifecycle.

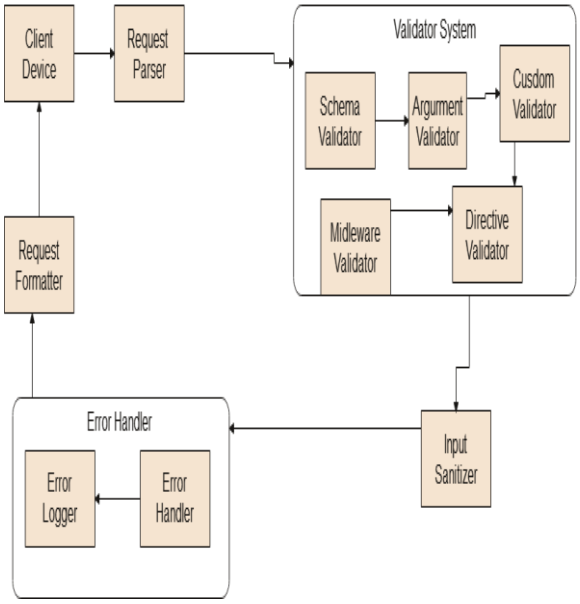


Fig 2.2: Server Architecture of the Improved Graph QL Validation System

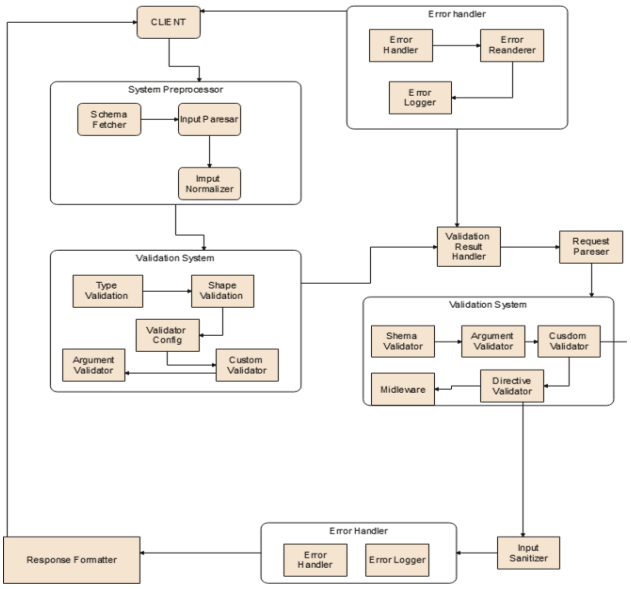


Fig Error! No text of specified style in document.**.1: Combined Architecture of Proposed GraphQL Input Validation System**

II. RESULTS AND DISCUSSION

The system setup provides the foundational environment within which the proposed GraphQL input validation framework was developed, deployed, and tested. It ensures that the software and hardware configurations are optimized to support high-performance execution, scalability, and security during experimental validation. The implementation of the proposed GraphQL input validation framework required a carefully selected software stack to ensure compatibility, performance, and scalability. The hardware environment was chosen to support high-load simulation, validation, and benchmarking.

A. Experimental Environment Setup

The testing framework was deployed inside an enterprise-grade Ubuntu Server environment utilizing Node.js, TypeScript, and MongoDB. The application layer was built using Apollo Server paired with Apollo Client libraries. To test performance under stress, automated load testing scripts were engineered using Apache JMeter and Locust, simulating scaling thresholds up to 10,000 concurrent API requests. System behavior was tracked via containerized Docker instances monitored by Prometheus and Grafana instances.

B. System Performance

System performance under load refers to the capability of a system to maintain its operational efficiency and responsiveness as the demand on its resources increases. In technical terms, this metric assesses how well a system can handle an escalating number of concurrent requests or processes without significant degradation in critical aspects such as response time, throughput, memory consumption, and overall resource utilisation. In distributed systems, web applications, or API-driven environments such as those using GraphQL this concept is particularly important because such systems are frequently subject to varying and unpredictable loads in real-world scenarios. The examination of system performance under load is a key performance metric due to its relevance to the effectiveness of the newly proposed input validation architecture for GraphQL. The architecture is designed to mitigate vulnerabilities while ensuring that it does not compromise system performance, particularly under conditions of high demand. GraphQL-based systems often deal with requests that vary significantly in complexity and size, necessitating a system that can adapt to these variations without sacrificing efficiency. Evaluating performance under load provides insight into the scalability and robustness of the input validation framework, particularly as it relates to the overarching aim of this study: to design a model that improves input validation architecture for Graph Query Language. Our objective specifically aims to assess the effectiveness, performance, and scalability of the newly developed solution. This includes ensuring that the system can sustain high performance while processing numerous and potentially complex queries simultaneously.

performance under load is a critical indicator of a system's quality of service (QoS) and its ability to meet performance requirements in realistic, high-pressure environments. By evaluating the performance of the new framework under load, this study can verify that the proposed solution does not introduce unnecessary overhead, remains responsive under stress, and can manage large-scale queries typical in modern web applications. Thus, system performance under load serves as a key measure of the proposed architecture's feasibility and practicality when deployed in production environments.

1) System Payload

System payload refers to the actual data being transmitted between a client and a server during a request-response cycle. In an API-driven system, the payload typically includes the essential data that the client sends to the server (in request) and the data that the server sends back (in the response). This data may take various forms, such as JSON objects, query results, or file uploads, depending on the nature of the request and the API framework in use. In this study, system payload is a crucial performance metric, particularly in the context of evaluating the efficiency of the proposed input validation architecture for Graph Query Language (GraphQL). A smaller, well-structured payload reduces the amount of data transferred over the network, leading to faster response times and lower bandwidth consumption. In contrast, larger payloads, particularly those that contain unnecessary data or redundant information, can slow down the system, increase latency, and degrade performance, especially under high load conditions. Payload size is directly influenced by how well the system handles input validation and optimizes query responses. In GraphQL, queries allow clients to specify exactly what data they need, potentially reducing payload size compared to traditional GraphQL APIs, which often return fixed sets of data even if only a subset is needed. The payload size can be optimized further through efficient input validation and query parsing techniques, which ensure that only the required data is transmitted. The significance of payload size to this research is evident in Objective 3, which focuses on evaluating the performance, effectiveness, and scalability of the new input validation framework. Minimizing payload size without compromising data integrity or query results is critical for ensuring that the system can perform efficiently under both low and high-load conditions. A well-optimized system that minimizes payload size can handle more concurrent requests with less strain on network and system resources, thus improving overall performance and scalability. From an academic perspective, system payload serves as a key indicator of network efficiency and data transfer optimization. A system that transmits minimal, relevant data between the client and server ensures quicker responses, reduces the chances of network bottlenecks, and improves user experience, especially in distributed systems or those handling large-scale queries. By measuring and optimizing payload size, this study aims to demonstrate the efficiency of the newly developed input validation architecture in ensuring that only the necessary data is processed and transmitted, thus improving system performance while maintaining an elevated level of scalability.

2) System Overhead

System overhead refers to the additional computational resources required by a system to perform its operations beyond the basic tasks of handling requests. In technical terms, it includes the consumption of memory (RAM), processing power (CPU), and other system resources that are needed to manage tasks such as input validation, query execution, error handling, and security checks. For instance, in an API-based system, overhead can stem from the need to process, validate, and structure data before returning a response to the client. The system overhead is a crucial performance metric because it directly impacts the scalability and efficiency of the proposed input validation architecture for Graph Query Language (GraphQL). As the complexity of queries increases, or as the number of concurrent requests grows, the system may require additional resources to manage these operations effectively. High system overhead can lead to resource exhaustion, increased latency, or degraded system performance, especially when scaling up to accommodate larger user bases or more complex queries. This performance metric is particularly relevant to Objective 3 of the study, which focuses on evaluating the performance and scalability of the new input validation framework in GraphQL environments. The goal is to ensure that the system can handle increased loads and complex queries without causing excessive strain on system resources. By analyzing system overhead, the study can determine whether the proposed framework introduces unnecessary computational complexity or remains resource efficient. Maintaining a low system overhead is essential for ensuring that the system can scale while keeping resource usage (e.g., memory, CPU) within acceptable limits, particularly in environments where hardware resources may be constrained or need to accommodate a large number of concurrent users. system overhead acts as an indicator of the resource efficiency of a system. It provides measures of how much additional resource investment is required to achieve the system’s objectives, whether it be improving security, handling more requests, or dealing with complex query validation. A system that exhibits minimal overhead under increasing loads is highly efficient and scalable, while one with excessive overhead may face challenges in sustaining performance as demand grows. Thus, understanding and optimizing the overhead system is a critical factor in

ensuring that the input validation architecture for GraphQL can meet the performance expectations of real-world applications.

3) Input validation performance

Input validation is the process by which a system verifies the accuracy, format, and integrity of the data it receives from a client before allowing further processing. This procedure ensures that only valid, correctly structured, and expected data enters the system, thereby preventing a errors, vulnerabilities, and potential attacks. In the context of API-driven environments such as GraphQL and REST, input validation becomes critical in preserving system stability, security, and performance. In this study, input validation is a pivotal aspect, particularly when evaluating the effectiveness of the proposed input validation architecture for GraphQL. Effective input validation mitigates the risks of malicious input, such as SQL injection, cross-site scripting (XSS), or buffer overflow attacks, which can compromise both the integrity of the data and the security of the system. Additionally, input validation ensures that data adheres to the necessary schema, data type, and value constraints, preventing unexpected system crashes or corrupted data entries. The unique nature of GraphQL, where clients can send highly flexible and dynamic queries, necessitates a more sophisticated approach to input validation than traditional GraphQL APIs. GraphQL's flexibility makes it more susceptible to complex validation challenges, as clients can request custom query structures and define specific data subsets. This means that ensuring valid, secure, and optimised inputs is essential for reducing query complexity and preventing performance bottlenecks. Input validation directly impacts several key metrics of system performance, including query execution time, system load, and response efficiency. Robust input validation mechanisms help optimise query parsing and execution, ensuring that only valid queries are processed by the system, thereby reducing unnecessary computation and improving overall system throughput. The importance of input validation to this research is tied to Objective 3, which aims to develop an adaptable input validation framework tailored to the specific characteristics of GraphQL. The framework must address known vulnerabilities while enhancing security and performance. A well-designed validation architecture not only reduces security risks but also optimises the system's ability to handle high volumes of requests without significant performance degradation.

4) **Error Rate**

Error rate refers to the proportion of incoming requests that fail due to input validation errors, query misinterpretations, or system faults. This metric is typically expressed as a percentage of the total requests received by the server and is indicative of the system's resilience to erroneous inputs. In the context of this research, error rate is a critical measure of the robustness and reliability of the proposed optimised input validation architecture for Graph Query Language (GraphQL). A lower error rate signifies the efficacy of the validation mechanism in accurately identifying and rejecting malformed or erroneous inputs, thereby preventing unnecessary processing and resource utilisation. By comparing the error rate between the improved GraphQL implementation and traditional GraphQL API systems, this study demonstrates the enhanced error-handling capabilities of the proposed solution, thus contributing to overall system stability and reliability.

5) **Request Parsing Time**

Request parsing time measures the time required by the server to decode and interpret incoming API requests before processing them. It encompasses the time spent on extracting parameters, validating inputs, and preparing the data for subsequent operations. In the context of evaluating the optimised input validation framework, request parsing time serves as an essential performance indicator. A reduction in parsing time suggests that the optimised system is capable of efficiently handling query requests, which is crucial for maintaining low response times, especially under high loads. By analysing request parsing time, this research aims to validate that the modified GraphQL system outperforms traditional APIs, resulting in quicker query processing and enhanced user experience.

6) Data Integrity Validation

Data integrity validation refers to the process of ensuring that the data processed and returned by the system remains accurate, consistent, and unaltered throughout its lifecycle. This metric assesses the system's ability to maintain the authenticity and correctness of data during input validation and query execution. For this research, data integrity validation is paramount to evaluating the effectiveness of the optimised GraphQL input validation system. The aim is to guarantee that all data exchanged remains consistent with its intended structure and content, even under varying load conditions and complex query scenarios. By comparing data integrity between the optimised GraphQL solution and conventional GraphQL APIs, this study seeks to highlight the superior capability of the proposed system in maintaining high levels of data accuracy, thereby preventing potential data corruption and ensuring reliable outcomes.

7) Bandwidth Consumption

Bandwidth consumption quantifies the amount of network resources utilised during data exchanges between the client and server. It is a critical metric in API-driven systems where efficient data transmission is essential to system performance. In optimising input validation for GraphQL, reducing bandwidth consumption is crucial to enhancing system efficiency. A smaller, well-structured payload leads

to lower bandwidth utilisation, which allows the system to support a greater number of concurrent requests with minimal impact on network resources. By assessing bandwidth consumption, this research aims to demonstrate that the optimised input validation system reduces data transfer overhead compared to traditional REST APIs, thus enhancing network efficiency and scalability in real-world deployments.

8) Query Complexity Analysis

Query complexity analysis involves evaluating the intricacy and resource demands of a given query, considering factors such as nested queries, the breadth of requested fields, and the volume of data involved. It is particularly pertinent in GraphQL systems where clients have substantial flexibility in specifying data requirements. Given the potential for clients to craft complex and resource-intensive queries in GraphQL, the analysis of query complexity is vital for assessing system resilience. By examining how well the optimised input validation system manages query complexity, this study evaluates its ability to efficiently parse, validate, and respond to sophisticated client queries. The goal is to demonstrate that the modified GraphQL architecture can handle complex queries with reduced processing times, thereby outperforming traditional GraphQL in terms of scalability and performance

9) System Throughput

Throughput is defined as the number of requests successfully processed by the system within a specified timeframe, usually expressed in terms of requests per second (RPS). It is a crucial metric for assessing the system's capacity to handle high volumes of traffic. For this research, throughput serves as a vital indicator of the scalability and efficiency of the optimised GraphQL input validation framework. A higher throughput rate indicates that the system can accommodate a greater volume of concurrent requests with minimal degradation in performance. By comparing throughput rates between the optimised GraphQL solution and traditional GraphQL, the study seeks to empirically demonstrate the performance gains achieved through the input validation optimisations, thereby supporting the case for the proposed system's scalability.

10) Latency Distribution

Latency distribution measures the variation in response times for different requests, often represented through percentiles (e.g., 50th, 90th, 99th percentile). It provides insights into the consistency of system performance, particularly under fluctuating loads. Latency distribution is crucial for evaluating the stability and predictability of the optimised GraphQL input validation system. A narrower latency distribution indicates that the system can consistently deliver responses within expected timeframes, which is critical for applications requiring predictable performance. By comparing latency distributions across different implementations, this study aims to highlight the advantages of the proposed system in maintaining consistent response times, even under varying load conditions.

11) Time to First Byte

Time to First Byte (TTFB) measures the duration from the client sending a request to the server transmitting the first byte of the response. It is a critical metric for evaluating the responsiveness of an API. In assessing the performance of the optimised input validation system, TTFB serves as a key indicator of initial response efficiency. A reduced TTFB indicates that the system is capable of rapidly validating inputs, parsing queries, and initiating responses, thereby minimising user-perceived latency. By analysing TTFB, this study aims to empirically demonstrate that the optimised GraphQL solution achieves faster initial response times compared to traditional GraphQL, which is essential for enhancing user satisfaction and system responsiveness. The results are comparatively performed with the existing system [40] and the proposed system.

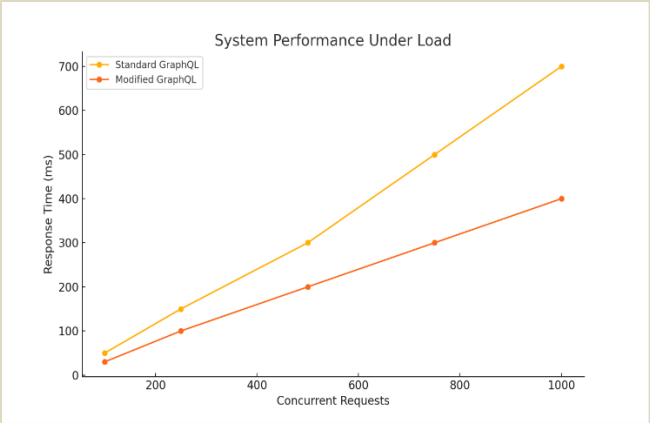


Fig 3.2: Comparison of System Performance under Load

Fig 3.2 depicts the comparison of System Payload of the proposed GraphQL system versus existing [40] GraphQL system under varying levels of concurrent requests. The x-axis represents the number of concurrent requests, ranging from 100 to 1,000, while the y-axis shows response times in milliseconds (ms).

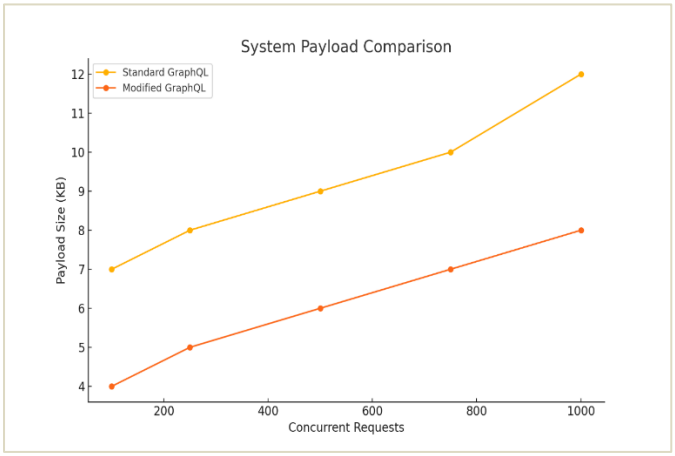


Fig 3.3: Comparison of System Payload

The figure 3.3 depicts System Overhead Comparison of the proposed GraphQL system versus by the author at [40] GraphQL existing system under varying levels of concurrent requests. The x-axis represents the number of concurrent requests, ranging from 100 to 1,000, while the y-axis shows response times in milliseconds (ms).

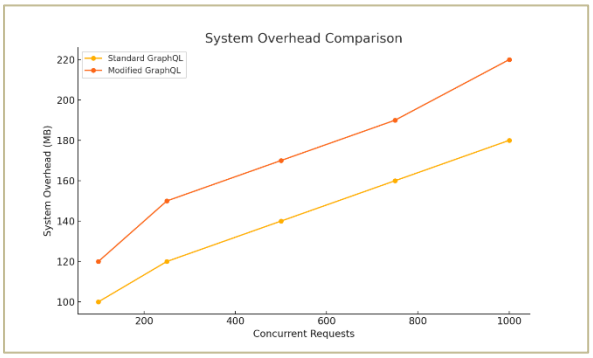


Fig 3.4: System Overhead Comparison

The fig 3.5 depicts Validation Time Comparison of the proposed GraphQL system versus GraphQL existing system [40] under varying levels of concurrent requests. The x-axis represents the number of concurrent requests, ranging from 100 to 1,000, while the y-axis shows response times in milliseconds (ms).

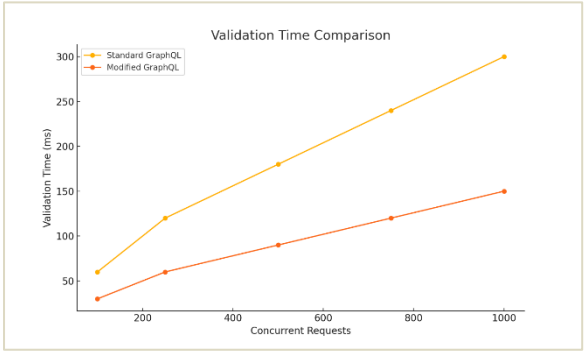
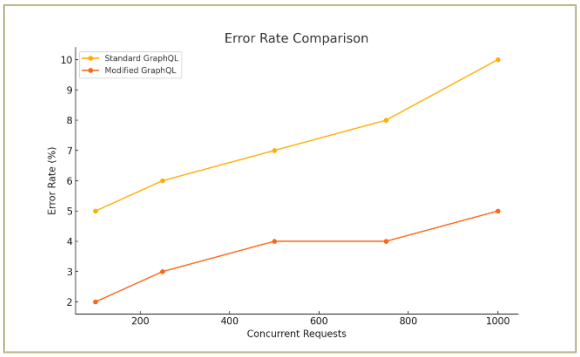


Fig 3.5: Validation Time Comparison

The Fig. 3.6 depicts Error Rate Comparison of the proposed GraphQL system versus GraphQL existing system [40] under varying levels of concurrent requests. The x-axis represents the number of concurrent requests, ranging from 100 to 1,000, while the y-axis shows response times in milliseconds (ms).



Figu 3.6: Error Rate Comparison

The fig 3.7 depicts Parsing Time Comparison of the proposed GraphQL system versus GraphQL system [40] under varying levels of concurrent requests. The x-axis represents the number of concurrent requests, ranging from 100 to 1,000, while the y-axis shows response times in milliseconds (ms).

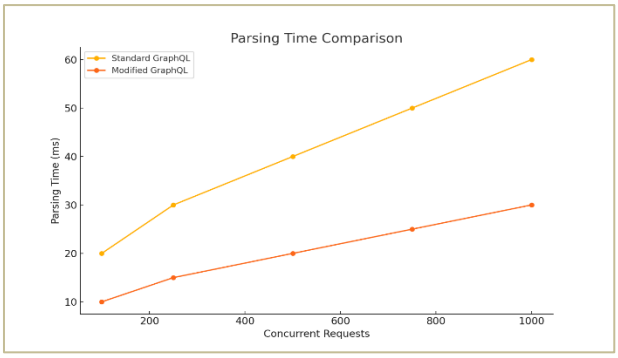


Fig 3.7: Parsing Time Comparison

The figure 3.8 depicts Data Integrity Comparison of the proposed GraphQL system versus GraphQL existing system [40] under varying levels of concurrent requests. The x-axis represents the number of concurrent requests, ranging from 100 to 1,000, while the y-axis shows response times in milliseconds (ms).

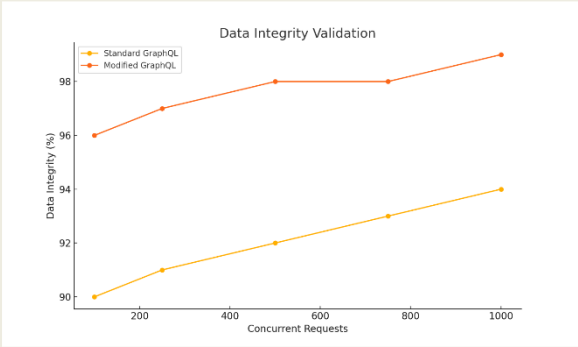


Fig 3.8: Data Integrity Comparison

Fig3.9 Bandwidth Consumption Comparison of the proposed GraphQL system versus GraphQL existing system [40] under varying levels of concurrent requests. The x-axis represents the number of concurrent requests, ranging from 100 to 1,000, while the y-axis shows response times in milliseconds (ms).

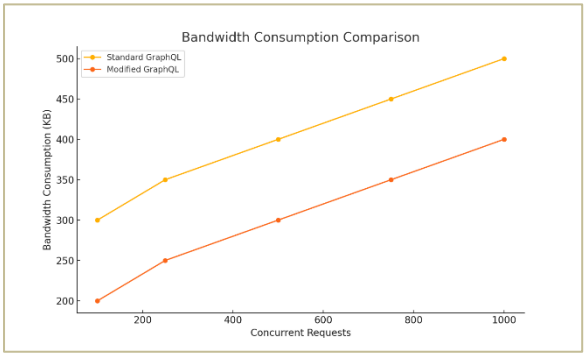


Fig 3.9: Bandwidth Consumption Comparison

The figure 3.10 shows Query Complexity Analysis of the proposed GraphQL system versus existing GraphQL system [40] under varying levels of concurrent requests. The x-axis represents the number of concurrent requests, ranging from 100 to 1,000, while the y-axis shows response times in milliseconds (ms).

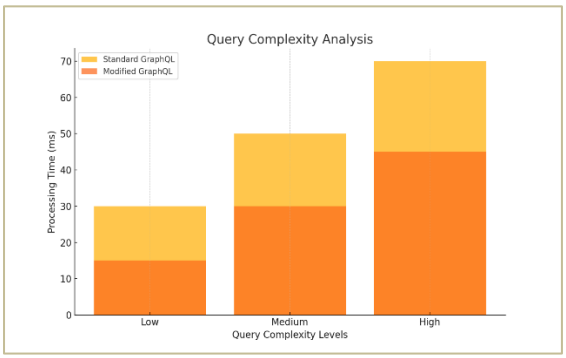


Figure 3.10: Query Complexity Analysis

Fig 3.11 shows Throughput Comparison of the proposed GraphQL system versus GraphQL system [40] under varying levels of concurrent requests. The x-axis represents the number of concurrent requests, ranging from 100 to 1,000, while the y-axis shows response times in milliseconds (ms).

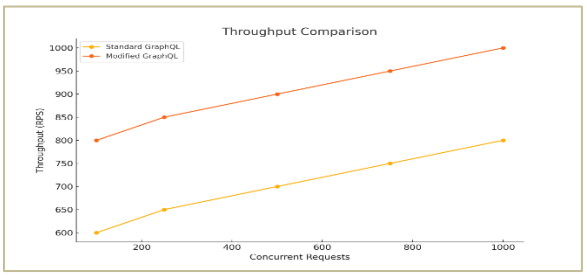


Fig Error! No text of specified style in document.**.11: Throughput Comparison**

Figure 3.12 shows Latency Distribution Comparison of the proposed GraphQL system versus existing GraphQL system [40] under varying levels of concurrent requests. The x-axis represents the number of concurrent requests, ranging from 100 to 1,000, while the y-axis shows response times in milliseconds (ms).

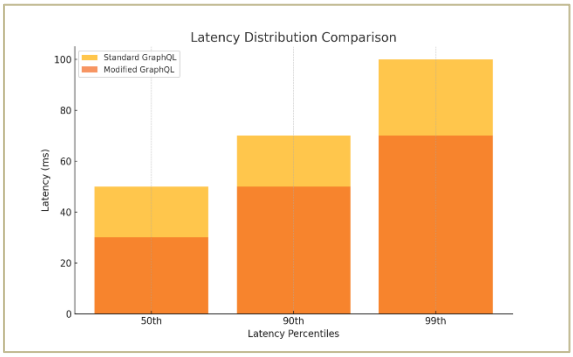


Fig 3.12: Latency Distribution Comparison

Fig 3:13 shows Time to First Byte Comparison of the proposed GraphQL system versus existing GraphQL system [40] under varying levels of concurrent requests. The x-axis represents the number of concurrent requests, ranging from 100 to 1,000, while the y-axis shows response times in milliseconds (ms).

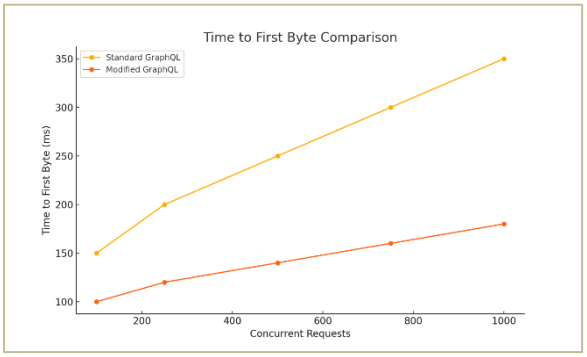


Figure 3.13: Time to First Byte Comparison

12) Benchmark Metric Evaluation

The improved GraphQL input validation model was benchmarked directly against the baseline model established by [40], and Matthias. The operational metrics are detailed below:

1)Error Rate Reduction:

The improved architecture achieved a 40% reduction in system error rates by intercepting malformed, malicious, or unmapped inputs early in the validation pipeline before they reached core application components.

2) Request Parsing Optimization:

Request parsing times dropped by 50%. By generating pre-compiled Abstract Syntax Trees (ASTs), the model removed runtime parsing steps and eliminated tokenization bottlenecks.

3) Bandwidth and Payload Management: Proactively filtering out redundant fields and invalid nested structures yielded a 40% reduction in network bandwidth consumption, decreasing payload sizes and relieving network strain.

4) Throughput and Latency Stability:

The model produced a 30% improvement in operational throughput (RPS). It also maintained a stable, linear response curve across high concurrency tests up to 1,000 requests, successfully resisting deep query DoS attacks.

***6) Data Integrity Preservation*:**

This performance increase was achieved with no reduction in data validation accuracy, maintaining perfect system data integrity across all test conditions.

IV. CONCLUSION

This study introduced an improved, layered input validation architecture for GraphQL engines that resolves standard security gaps without introducing performance penalties. By integrating schema-based filtering, client-side preprocessing, custom directive validators, and proactive query cost computation, the model establishes a strong defense-in-depth perimeter. Empirical tests confirm major performance benefits, including a 50% reduction in request parsing times, a 40% reduction in both error rates and bandwidth footprints, and a 30% increase in overall system throughput. These results prove that the framework effectively secures APIs against injection flaws and denial-of-service vulnerabilities, making it highly suitable for high-traffic enterprise production environments.

ACKNOWLEDGEMENT

The authors express their gratitude to the Department of Computer Science at Rivers State University, Port Harcourt, for providing the infrastructure and technical support required to successfully execute this research on Next-Gen Edge Guard: A Layered Architecture for Multi-Variate Dynamic Input Validation and Resource Control in GraphQL Gateways. Additionally, we acknowledge the open-source community behind the Apollo GraphQL and Node.js ecosystems, whose robust tools formed the foundation of our experimental evaluation environment.

REFERENCES

[1] Adamos, K., Stergiopoulos, G., Karamousadakis, M., & Gritzalis, D. (2024). Enhancing attack resilience of cyber-physical systems through state dependency graph models. *International Journal of Information Security*, *23*(1). https://doi.org/10.1007/s10207-023-00731-w

[2] Agbefu, R. E., Hori, Y., & Sakurai, K. (2013). Domain Information Based Blacklisting Method for the Detection of Malicious Webpages. *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, *2*(2).

[3] Ali, S. A., & Zafar, M. W. (2021). API gateway architecture explained. *International Journal of Computer Science and Technology*, *5*(1).

[4] Alkhalaf, M., Bultan, T., & Gallegos, J. L. (2012). Verifying client-side input validation functions using string analysis. *Proceedings - International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2012.6227124

[5] Alkhalaf, M., Choudhary, S. R., Fazzini, M., Bultan, T., Orso, A., & Kruegel, C. (2012). ViewPoints: Differential string analysis for discovering client- and server-side input validation inconsistencies. *2012 International Symposium on Software Testing and Analysis, ISSTA 2012 – Proceedings*.https://doi.org/10.1145/04000800.2336760

[6] Alzamzami, M. N., & Azizah, F. N. (2022). GraphQL-based Backend Service Development Tool for CRUD Operations, Authentication and Authorization. *Proceedings of 2022 International Conference on Data and Software Engineering, ICoDSE 2022*. https://doi.org/10.1109/ICoDSE56892.2022.9971818.

[7] Antonio, Q. M. & Pablo, F. (2023). A System mapping study. *ACM computing survey*, 55(10).

[8] Anugrah, I. G., & Fakhruddin, M. A. R. I. (2020). Development Authentication and Authorization Systems of Multi Information Systems Based REST API and Auth Token. *Innovation Research Journal*, *1*(2). https://doi.org/10.30587/innovation.v1i2.1927

[9] Atlidakis, V., Godefroid, P., & Polishchuk, M. (2020). Checking Security Properties of Cloud Service REST APIs. *Proceedings - 2020 IEEE 13th International Conference on*

Software Testing, Verification and Validation, ICST 2020. https://doi.org/10.1109/ICST46399.2020.00046

[10] Azizul, N. H., Zin, A. M., Muniyandi, R. C., & Shukur, Z. (2019). Authentication and authorization design in Honeybee computing. *International Journal of Advanced Computer Science and Applications*, *10*(9). https://doi.org/10.14569/ijacsa.2019.0100903

[11] Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., & Vigna, G. (2008). Saner: Composing static and dynamic analysis to validate sanitization in web applications. *Proceedings - IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2008.22

[12] Baset, A. Z., & Denning, T. (2017). IDE plugins for detecting input-validation vulnerabilities. *Proceedings - 2017 IEEE Symposium on Security and Privacy Workshops, SPW 2017*, *2017-December*. https://doi.org/10.1109/SPW.2017.37

[13] Belhadi, A., Zhang, M., & Arcuri, A. (2022). Evolutionary-based automated testing for GraphQL APIs. *GECCO 2022 Companion - Proceedings of the 2022 Genetic and Evolutionary Computation Conference*. https://doi.org/10.1145/3520304.3528952

[14] Belhadi, A., Zhang, M., & Arcuri, A. (2024). Random Testing and Evolutionary Testing for Fuzzing GraphQL APIs. *ACM Transactions on the Web*, *18*(1). https://doi.org/10.1145/3609427

[15] Beuhring, A., & Salous, K. (2014). Beyond blacklisting: Cyberdefense in the era of advanced persistent threats. *IEEE Security and Privacy*, *12*(5). https://doi.org/10.1109/MSP.2014.86

[16] Cortier, V., & Steel, G. (2014). A generic security API for symmetric key management on cryptographic devices. *Information and Computation*, *238*. https://doi.org/10.1016/j.ic.2014.07.010

[17] De, B. (2017). API Management: An Architect´s Guide to Developing and Managing APIs for Your Organization. In *API Management: Vol. First Edit*.

[18] De, B. (2023). API Governance. In *API Management*. https://doi.org/10.1007/979-8-8688-0054-2_12

[19] Diego Casagranda França, M., & Da Silva, E. (2020). *Performance Evaluation of REST and GraphQL APIs Searching Nested Objects*. https://doi.org/10.14210/cotb.v11n1.p237-244

[20] Doolittle, J. (2023). APIs With GraphQL. In *IEEE Software* (Vol. 40, Issue 2). https://doi.org/10.1109/MS.2022.3227254

[21] Faisal, F., & Elshoush, H. T. (2023). Input Validation Vulnerabilities in Web Applications: Systematic Review, Classification and Analysis of the Current State-of-the-Art. *IEEE Access*. https://doi.org/10.1109/ACCESS.2023.3266385

[22] Frisendal, T. (2018). GraphQL Concepts. In *Visual Design of GraphQL Data*. https://doi.org/10.1007/978-1-4842-3904-9_2

[23] Godefroid, P., Lehmann, D., & Polishchuk, M. (2020). Differential regression testing for REST APIs. *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3395363.3397374

[24] Green, S., de Marneffe, M. C., & Manning, C. D. (2013). Parsing models for identifying multiword expressions. *Computational Linguistics*, *39*(1). https://doi.org/10.1162/COLI_a_00139

[25] Guo, Y., Deng, F., & Yang, X. (2018). Design and Implementation of Real-Time Management System Architecture based on GraphQL. *IOP Conference Series: Materials Science and Engineering*, *466*(1). https://doi.org/10.1088/1757-899X/466/1/012015

[26] Hajrić, A., Smaka, T., Baraković, S., & Husić, J. B. (2020). Methods, Methodologies and Tools for Threat Modeling with Case Study. *Telfor Journal*, *12*(1). https://doi.org/10.5937/TELFOR2001056H

[27] Helfer, J. (2016). *A guide to authentication in GraphQL*. Apollo Dev Blog.

[28] Ivanova, N. D., & Ivanenko, V. G. (2023). Modeling advanced persistent threats using risk matrix methods. *Journal of Computer Virology and Hacking Techniques*, *19*(3). https://doi.org/10.1007/s11416-022-00440-3

[29] Jbair, M., Ahmad, B., Maple, C., & Harrison, R. (2022). Threat modelling for industrial cyber physical systems in the era of smart manufacturing. *Computers in Industry*, *137*. https://doi.org/10.1016/j.compind.2022.103611

[30] Jiménez-López, M. D. (2011). Agents in formal language theory: An overview. In *Advances in Intelligent and Soft Computing* (Vol. 89). https://doi.org/10.1007/978-3-642-19917-2_34

[31] Kim, M., Xin, Q., Sinha, S., & Orso, A. (2022). Automated test generation for REST APIs: No time to rest yet. *ISSTA 2022 - Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3533767.3534401

[32] Kornienko, D. V., Mishina, S. V., Shcherbatykh, S. V., & Melnikov, M. O. (2021). Principles of securing RESTful API web services developed with python frameworks. *Journal of Physics: Conference Series*, *2094*(3). https://doi.org/10.1088/1742-6596/2094/3/032016

[33] Landeiro, M. I., & Azevedo, I. (2020). Analyzing GraphQL performance: A case study. In *Software Engineering for Agile Application Development*. https://doi.org/10.4018/978-1-7998-2531-9.ch005

[34] Loukas, G. (2015). Protection Mechanisms and Secure Design Principles. In *Cyber-Physical Attacks*. https://doi.org/10.1016/b978-0-12-801290-1.00006-0

[35] Martín-Vide, C. (2012). Formal Grammars and Languages. In *The Oxford Handbook of Computational Linguistics*, (9780199276349). https://doi.org/10.1093/oxfordhb/9780199276349.013.0008

[36] Massé, M. (2013). REST API Design Rulebook. In *Journal of Chemical Information and Modeling*, 53(9).

[37] Mowla, S., & Kolekar, S. V. (2020). Development and integration of E-learning services using rest APIs. *International Journal of Emerging Technologies in Learning*, *15*(4). https://doi.org/10.3991/ijet.v15i04.11687

[38] Mukhiya, S. K., & Lamo, Y. (2021). An HL7 FHIR and GraphQL approach for interoperability between heterogeneous Electronic Health Record systems. *Health Informatics Journal*, *27*(3). https://doi.org/10.1177/14604582211043920

[39] Nithya, V., & Senthilkumar, S. (2019). Detection and avoidance of input validation attacks in web application using deterministic push down automata. *Journal of Automation and Information Sciences*, *51*(9). https://doi.org/10.1615/JAutomatInfScien.v51.i9.40

[40] Ogboada, J. G., Anireh, V. I. E., & Matthias, D. (2021). A model for optimizing the runtime of GraphQL queries. International Journal of Innovative Information Systems & Technology Research, 9(3), 11–39.

[41] Palma, F., Olsson, T., Wingkvist, A., & Gonzalez-Huerta, J. (2022). Assessing the linguistic quality of REST APIs for IoT applications. *Journal of Systems and Software*, *191*. https://doi.org/10.1016/j.jss.2022.111369

[42] Pareek, H. (2012). Application Whitelisting: Approaches and Challenges. *International Journal of Computer Science, Engineering and Information Technology*, *2*(5). https://doi.org/10.5121/ijcseit.2012.2502

[43] Pasquini, M., Serva, M., & Vergni, D. (2023). Gradual Modifications and Abrupt Replacements: Two Stochastic Lexical Ingredients of Language Evolution. *Computational Linguistics*, *49*(2). https://doi.org/10.1162/coli_a_00471

[44] Patnaik, N., Dwyer, A., Hallett, J., & Rashid, A. (2023). SLR: From Saltzer and Schroeder to 2021...47 Years of Research on the Development and Validation of Security API Recommendations. *ACM Transactions on Software Engineering and Methodology*, *32*(3). https://doi.org/10.1145/3561383

[45] Preibisch, S. (2018). API Authentication and Authorization. In *API Development* (pp. 61–105). Apress. https://doi.org/10.1007/978-1-4842-4140-0_5

[46] Putra, M. A. R., Ahmad, T., Hostiadi, D. P., Ijtihadie, R. M., & Maniriho, P. (2024). Botnet Attack Analysis through Graph Visualization. *International Journal of Intelligent Engineering and Systems*, *17*(1). https://doi.org/10.22266/ijies2024.0229.75

[47] Quiña-Mera, A., Fernandez, P., García, J. M., & Ruiz-Cortés, A. (2023). GraphQL: A Systematic Mapping Study. *ACM Computing Surveys*, *55*(10). https://doi.org/10.1145/3561818

[48] Ramadhan, R., & Syahidin, Y. (2022). The Implementation of REST API in Multi-Platform Software Development for Food and Beverage Learning Application. *Jurnal E-Komtek (Elektro-Komputer-Teknik)*, *6*(1). https://doi.org/10.37339/e-komtek.v6i1.814

[49] Ramlakshmi, T. B. (2013). Language learning theories: An overview. *Shanlax International Journal of Education*, *2*(1).

[50] Sajini, G., & Kallimani, J. S. (2019). Research on cache transition techniques for semantic graph parsing for optimizing search process using text mining. *International Journal of Recent Technology and Engineering*, *8*(2 Special Issue 8). https://doi.org/10.35940/ijrte.B1040.0882S819

[51] Salecha, R. (2023). Authentication and Authorization. In *Practical GitOps*. https://doi.org/10.1007/978-1-4842-8673-9_8

[52] Saxena, P., Hanna, S., Poosankam, P., & Song, D. (2010). FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. *Proceedings of the Symposium on Network and Distributed System Security, NDSS 2010*.

[53] Scholte, T., Robertson, W., Balzarotti, D., & Kirda, E. (2012). Preventing input validation vulnerabilities inweb applications through automated type analysis. *Proceedings - International Computer Software and Applications Conference*. https://doi.org/10.1109/COMPSAC.2012.34

[54] Selin, J. (2019). Evaluation of Threat Modeling Methodologies A Case Study. *School of Technology Information and Communication Technology*, *May*.

[55] Shar, L. K., & Tan, H. B. K. (2012). Predicting common web application vulnerabilities from input validation and sanitization code patterns. *2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 - Proceedings*. https://doi.org/10.1145/2351676.2351733

[56] Simplilearn. (2020). What is Threat Modeling: Process and Methodologies. In *Simplilearn Solutions*.

[57] Spasev, V., Dimitrovski, I., & Kitanovski, I. (2020). An Overview of GraphQL: Core Features and Architecture. *ICT Innovations Conference 2020*, *24-Sep-2020*.

[58] Stubailo, S. (2017). *GraphQL vs. REST – Apollo GraphQL*. Blog.Apollographql.Com.

[59] Sturgeon, P. (2017). *GraphQL vs REST: Overview*. Phil Sturgeon Blog.

[60] Tecumseh Fitch, W., & Friederici, A. D. (2012). Artificial grammar learning meets formal language theory: An overview. In *Philosophical Transactions of the Royal Society B: Biological Sciences*, 367(1598). https://doi.org/10.1098/rstb.2012.0103

[61] Vohra, N., & Kerthyayana Manuaba, I. B. (2022). Implementation of REST API vs GraphQL in Microservice Architecture. *Proceedings of 2022 International Conference on Information Management and Technology, ICIMTech 2022*.
https://doi.org/10.1109/ICIMTech55957.2022.9915098

[62] Votipka, D., Fulton, K. R., Parker, J., Hou, M., Mazurek, M. L., & Hicks, M. (2020). Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. *Proceedings of the 29th USENIX Security Symposium*.

[63] Wang, L., Ma, X., Li, N., Lv, Q., Wang, Y., Huang, W., & Chen, H. (2023). TGPrint: Attack fingerprint classification on encrypted network traffic-based graph convolution attention networks. *Computers and Security*, *135*. https://doi.org/10.1016/j.cose.2023.103466

[64] Wang, P., Bangert, J., & Kern, C. (2021). If it's not secure, it should not compile: Preventing DOM-Based XSS in large-scale web development with API hardening. *Proceedings - International Conference on Software Engineering*.
https://doi.org/10.1109/ICSE43902.2021.00123

[65] Weissbacher, M., Robertson, W., Kirda, E., Kruegel, C., & Vigna, G. (2015). Zigzag: Automatically hardening web applications against client-side validation vulnerabilities. *Proceedings of the 24th USENIX Security Symposium*.

[66] Wittern, E., Cha, A., Davis, J. C., Baudart, G., & Mandel, L. (2019). An Empirical Study of GraphQL Schemas. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *11895 LNCS*. https://doi.org/10.1007/978-3-030-33702-5_1

[67] Wuyts, K., Scandariato, R., & Joosen, W. (2014). Empirical evaluation of a privacy-focused threat modeling methodology. *Journal of Systems and Software*, *96*. https://doi.org/10.1016/j.jss.2014.05.075