International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V3I3P112
Eureka Vision Publication | Volume 3, Issue 3, 112-121, 2022

Original Article

Mitigating Algorithmic Complexity Attacks in Federated GraphQL Architectures: A Depth-Bounded Semantic Rate Limiting Approach for Open Banking

Anvesh Katipelly[1], Narendra Kumar Kuntamukkala[2]
[1]Senior Software Engineer PayPal, Texas, USA.
[2]Senior Software Developer, Citi bank, Farmers Branch, TX.

Abstract - This swift financial services digitization has given rise to the open banking ecosystems which are powered by the contemporary API architecture. GraphQL has become very popular among them since it is flexible, data retrieval can be made efficient, and integrating multiple microservices into one unity can be achieved. The additional advantages of federated GraphQL are that such architectures support the use of distributed compositional services, such that independent development teams create scalable services that interoperate well with a single API gateway. Nevertheless, the security issues that are posed by this architectural paradigm are new, especially by the attacks associated with algorithmic complexity. The attacks are possible because the computational cost of deeply nested or semantically expensive GraphQL queries can be used to exhaust server resources in denial-of-service conditions. The difference between algorithmic complexity attacks and traditional volumetric attacks is that the attacks are not based on the volume of request. Rather, attackers construct a small set of queries that invoke computations which are computationally expensive like recursive field resolutions, cross-services joins, and graph deep traversal. Such attacks are more harmful in federated GraphQL environments deployed in open banking settings since the query can spread to several back-end services, increasing the computing costs. As a result, the few malicious queries can slow down the performance of the systems and undermine the availability of services. The current mitigation measures usually are based on either fixed query depths, query cost metrics, or conventional rate control methods. These strategies offer certain defence, but they are limited in a number of ways. The use of static depth limiting is known to block legitimate complex queries may be needed in financial analytics. Systems of query cost estimation are challenging to tune on a microservice distributed system. Traditional rate limiting schemes are more concerned with the frequency of requests as opposed to the computation complexity. Due to this, attackers are able to compromise such defenses by making low-frequency, but Poisson queries. In order to overcome these challenges, the current paper suggests an innovative conceptual security framework, namely, Depth-Bounded Semantic Rate Limiting (DBSRL). The suggested approach integrates the analysis of the structure of queries with the semantic interpretation of query implementation cost to dynamically manage API access. Differing with the traditional methods where the syntactic depth is only used, DBSRL considers query depth and semantic complexity based on resolver execution patterns and service dependencies.

Integrating these metrics in a dynamic rate limiting mechanism can enable the system to identify and partially mitigate to attack by algorithmic complexity as well as offer acceptable performance to end-users. The suggested framework works in three processes. To enable the data transfer, the GraphQL gateway interprets the queries received with the help of a structural parser which does compute the depth and breadth of the query tree. Second, a semantic analyzer approximates computational cost using resolver dependencies, past execution latency and invocation patterns of cross services within the federated architecture. Third, a dynamically rate limiting engine applies thresholds which change with the load of the system and user behaviour restrictions and stops excessive consumption of computational resources. The effectiveness of the proposed approach was tested through experiments done in simulated open banking microservices in a federated GraphQL environment. The metrics used in the evaluation are the query processing latency, the system throughput, CPU utilization and the rate of attack mitigation. Experiments prove that the given DBSRL mechanism mitigates the effects of the algorithm complexity attacks dramatically without affecting the performance under normal workload. Moreover, the suggested technique is more accurate in detection than traditional query depth limitation techniques. The semantic cost estimation can be introduced into the framework to differentiate between legitimate complex queries and malicious queries that aim to obtain unreasonable levels of computational workload. This feature would be quite essential in open banking platforms where authentic applications usually demand queries in multi-services when trying to aggregate accounts, transaction analytics, and financial reporting. This paper has threefold contributions. The first one provides an in-depth assessment of vulnerabilities in an algorithmic complexity of federated GraphQL systems by open banking systems. Second, it also presents the Depth-Bounded Semantic Rate Limiting framework which combines structural and semantic query analysis enabling better attack mitigation. Third, it offers empirical testing that proves the efficiency of the method suggested in terms of improving API security without impacting on the system performance. The findings show that semantic complexity assessment combined with adaptive rate limiting is a viable and scalable approach to securing the current financial services delivered via GraphQL. This study adds to the constantly

expanding topic of API security and sentences a strong defense model against open banking infrastructures with computational denial-of-service attacks.

Keywords - GraphQL Security, Semantic Rate Limiting, Algorithmic DoS Mitigation, Query Complexity Analysis, Federated Schema Stitching, Recursive Depth Bounding, Introspection Defense, Token Bucket Algorithms, Abstract Syntax Tree (AST) Parsing, Open Banking Standards, API Gateway Governance, Resource Exhaustion Prevention, Declarative Data Fetching, Distributed Denial of Service (DDoS),Angular Framework, Component-Based Architecture, Single-Page Applications (SPA), TypeScript, Dependency Injection, Reactive Programming, RxJS, Modular Frontend Architecture, Client-Side Rendering, Web Application Development, Cloud Infrastructure Automation, CI/CD Pipeline Engineering, Infrastructure as Code (Terraform/Bicep), Kubernetes & Container Orchestration, Cloud Security & DevSecOps, API Gateway Governance & Zero Trust, Monitoring, Observability & SRE Practices, Distributed Systems Reliability Engineering, Scalable Microservices Architecture, Identity & Access Management (IAM), Cost Optimization & Cloud Governance, Service Mesh & Network Policies, Secrets Management & Compliance Automation, Automated Deployment & Release Management, High Availability & Disaster Recovery Engineering.

1. Introduction

1.1. Background

The financial technology sector has grown tremendously since the advent of open banking that allows banks to share data safely with a third-party provider of services by standardized applications programming interfaces (APIs). These APIs enable various financial systems to exchange information effectively whilst being highly secure with a high level of compliance with regulations. [1,2] Historically, a variety of open banking platforms used to use REST-based APIs to support structured points of access to financial data. Nonetheless, with the requirements of the contemporary applications being more flexible, efficient data retrieval, and the possibility to combine the information presented by multiple microservices together, GraphQL has become a more favored API technology. GraphQL enables clients to only request the data that they require, preventing unwanted data transmission and enhancing the performance. It also allows schema-based data modeling which allows multiple distributed services to be integrated into a single API interface. Federated GraphQL is a popular schema in large-scale financial platforms, with each microservice providing individual schemas assembled at a central gateway into a global schema. This architecture is better in terms of modularity, scalability and its interface to external applications is seamless. Regardless of such benefits, GraphQL presents its own security issues based on flexible query representation. In contrast to REST APIs where the server has pre-defined endpoints, GraphQL permits a customer to design complex and deeply-nested queries that cross relationships in the data structure. This flexibility can be abused by the attackers who can work around malicious queries that need a lot of computation, database operations and cross-service communication. The first threat is that of the algorithmic complexity attack where instead of very many queries, the attacker sends only one query which is very complicated and thus consumes too much system resources. In federated systems, such queries may be spread over many microservices causing a massive crippling of computational overhead. Weaknesses in open banking systems in particular are due to the fact that financial apps often have complicated associations among accounts, transactions, payments, and customer information. Although legitimate applications will wear a multi-level query to provide financial insights in many cases, it is not an easy task to understand the difference between a legitimate complex query and an ill intent request. Thus, it needs effective security measures that help administrative complex queries and retain the flexibility and efficiency offered by GraphQL.

1.2. Security Challenges in Federated GraphQL

Federated graphql designs enable the design to be more flexible and scalable by integrating several microservices into one unified API. Nonetheless, such a distributed architecture exposes the system to some security threats which may impact system performance and reliability. [3,4] Given that GraphQL enables the clients to assemble extremely client-customized queries, the attacker may use some features of the system to create workloads that are excessively large. Deep query nesting, resolver chaining, spreading query cross-service, and resource consumption attacks are some of the security threats of federated spreads in GraphQL environments.

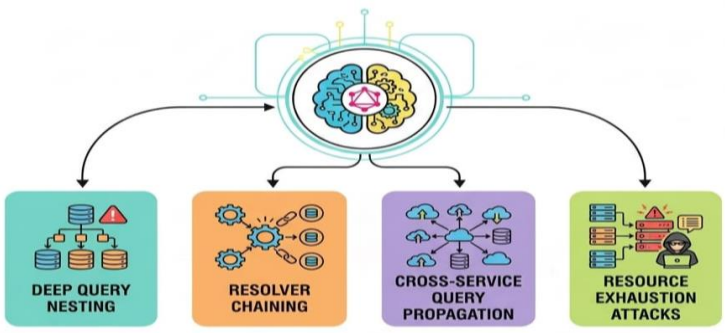

Figure 1. Security Challenges in Federated GraphQL

1.2.1. Deep Query Nesting

Deep query nesting takes place when a client builds a GraphQL object of nested fields in more than one way. Any level of nesting involves additional processing since the server needs to solve the relationship between various data objects. In complicated schemas, highly nested queries can cause numerous database accesses and computations. Attackers can deliberately craft queries that nest too exorbitantly to place a greater load on the processing burden on the server. This may result in long response times, greater use of the CPU and maybe denial-of-service situations when the system is unable to cope with the computing load.

1.2.2. Resolver Chaining

Resolver Chaining This occurs in that the execution of one resolver function is reached in such a way that it causes other resolver functions to be executed in a cascading manner. GraphQL uses resolvers to fetch data of certain fields in a query. In the event of requesting several fields, resolvers can rely on the output of other resolvers forming a chain of operations. Although such mechanism facilitates the process of efficient retrieval of data during valid queries, it may lead to performance problems in case too many resolvers are run during one request. Attacker However, it is possible to use resolver chaining to create queries that function to execute long chains of resolvers, causing the system to work much harder.

1.2.3. Cross-Service Query Propagation

In federated GraphQL architecture queries sent to the gateway can be shared between one or more backend services. Every microservice brings a part of the entire schema and a particular type of data. In case a complex query is issued, the query should be divided into smaller segments, and these segments should be sent to the services required. When the query has to be supported by a larger number of services, the request can be passed on to a number of systems before the response is provided. Attackers can use this behavior by forming queries, which use programs of multiple services at the same time, which raise the amount of network communication, time consumed, and system resources.

1.2.4. Resource Exhaustion Attacks

Resource exhaustion attacks are caused by malicious users, who make a query that consumes an unreasonable amount of processing resources like CPU usage, memory, database connections, etc. These attacks use few but very complex queries rarely done as opposed to the traditional distributed denial-of-service attacks which need a high number of requests. The effects of such attacks can be multiplied in federated GraphQL environments since several services can be used to execute the query. Otherwise, these queries may overwhelm the system, slow down the service availability and cause normal operations to come to a halt among honest users.

1.3. GraphQL Adoption in Open Banking

Introduction of GraphQL into open banking systems has become more common with the financial institutions David's in need of more elastic and streamlined methods of communicating through APIs. [5] Open banking, especially the one facilitated by the regulatory engine, i.e., the Payment Services Directive 2 (PSD2), demands banks and financial bodies to save secure access to financial details to authorized third-party provides. These policies promote the creation of similar APIs so that the fintech businesses, payment service providers, and other external applications can access banking services without compromising on the high security and regulatory integrity. Historically, most open banking platforms have leveraged REST based APIs to make financial data and services available. Although REST APIs offer systematic points of access and ensure stable communication, the interface tends to demand numerous requests to access other information connected with various resources. This may translate to a greater network overhead and an inefficient transfer of data particularly when the applications are required to collect complex financial information. GraphQL is a powerful alternative to REST being a flexible query-based model of data retrieval. GraphQL, unlike other multifix endpoints, enables clients to state precisely what information they are looking to get in a single query.

This method goes a long way in reducing network traffic since only the required information is relayed to the user therefore removing the issue of over-fishing information or under-fishing information. GraphQL has been used in open banking settings to provide a way to easily aggregate data across a number of services in a single call since applications often need to access related financial information, including accounts, transactions, payments, customer profiles, etc. The other benefit of GraphQL is that it supports changing and dynamic schema. Financial systems are usually constantly updated to either bring in new services, regulatory changes or to incorporate new financial products. GraphQL enables developers to add extensions to existing applications without introducing any type of it so that in the long-run, the APIs can be evolved. Moreover, with GraphQL it is easy to integrate distributed microservices, possible through federated architectures. In this type of systems, various services deal with particular details of data model and they add to a single schema, which is revealed by the central gateway. In general, the adoption of GraphQL in open banking enhances the level of performance and the extent of developer productivity alongside sustainable financial ecosystems. GraphQL can allow financial institutions to create contemporary interoperable platforms serving the rising needs of digital banking and fintech innovation by allowing flexible data querying and efficient integration of distributed services.

2. Literature Survey

2.1. GraphQL Security Research

GraphQL has been adopted by many developers to create flexible APIs since it enables clients to take the data they require. Nonetheless, some researches have identified that there are some possible security vulnerabilities to the flexible structure of its query. [6] The possibility that attackers can build very nesty queries or recursive field queries is one of the biggest concerns as this may put the server under a heavy load in terms of computation. Queries that have several nested relationships or recursive structures result in comparison and execution costs that produce growth of an exponentially increasing complexity which may cause a reduction in performance as well as denial-of-service conditions (DoS) situation. The researchers have thus highlighted query depth limiting, query complexity analysis and rigid validation as solutions to securing GraphQL servers against malicious or inefficient queries.

2.2. API Security in Open Banking

Open banking is the model that is heavily dependent on APIs, with their help, institutions can safely share data with third-party providers and customers. [7] Since these APIs contain sensitive financial data, there should be powerful security measures. The focus in studies in this field has been on embracing an effective authentication system such as OAuth 2.0 and stronger customer authentication (SCA) to verify the identity of users. Further, authorization controls are used to ensure that a particular service or data is only accessed by authorized applications. User rate limiting is also normally adopted in order to curb abuse as well as safeguard systems against extreme demands that may cripple the services. In combination, these security measures can ensure confidentiality, integrity and availability of open banking environments.

2.3. Query Cost Analysis Methods

Researchers have implemented several techniques to analyze the cost of queries in GraphQL systems to address the performance and security problems of the latter. [8] These approaches approximate the cost of calculation of using a query by the server. The GraphQL schema often represents each field in the GraphQL schema with a unique weight depending on its complexity, data accessibility needs, or resolver time. The cost of a query is then summed up as all the weights of requested fields and their nested structures. In case the estimated cost can be rejected or restricted in case it is greater than some defined limit. These models of cost estimation are useful in the prevention of resource exhaustion attacks and in order to keep the system stable in the event of heavy workloads.

2.4. Rate Limiting Techniques

Rate limiting is one of the strategies popular in regulating the number of requests that the clients are allowed to make to an API in a given time frame. [9] Conventional methods are Fixed Window, Token Bucket and Sliding Window. The Fixed Window technique puts a set amount of requests that may be placed within a specific fixed time frame say a minute but fails to consider the complexity of the requests. The Token Bucket is a method which allows to have bursts of traffic whereby it assigns tokens, which are consumed during the making of requests in that it offers more flexibility, but it does not yet take into account query cost or server workload. The Sliding Window method is one that enhances better traffic management since constant requests are kept in a moving time frame to provide better rate control. Nevertheless, these classical techniques primarily pay attention to the number of requests and not to the complexity of computing the request, which is not as effective in systems such as GraphQL where query cost can change drastically.

3. Methodology

3.1. System Architecture

The suggested framework is created to improve the security and efficiency of GraphQL APIs through analyzing and complex query requests and dynamically regulating request traffic. [10,11] The system architecture consists of a few interconnected modules that automatically work in harmony to process the incoming queries, approximate their cost to computation, and control the rate of requests to avoid overloading the system or malicious attacks.

Figure 2. System Architecture

3.1.1. Query Parser

The first element in the system architecture is the Query Parser which has the role of accepting and decoding incoming GraphQL queries. It transforms the query into a form that can be analyzed by the system, usually the Abstract Syntax Tree (AST), so as to better analyze the query content. Although it can be quite complex, the query structure can be used to identify fields, nested objects, arguments, and other relationships between various data elements by parsing the query structure. This organized representation is further transferred to the additional modules to undergo further analysis and processing.

3.1.2. Depth Analyzer

The Depth Analyzer looks at the query that is being analyzed to find out its degree of nesting and the complexity of the query. The access may cause overconsumption of resources and processes in GraphQL due to the fact that the level of nesting can result in an additional call of a resolver. The module identifies and compute the extent to which the query goes before the information is deemed safe. In case the query depth is above the maximum limit the system can either refuse the query or impose tougher requirements to eliminate possible denial-of-service attacks.

3.1.3. Semantic Cost Estimator

Semantic Cost Estimator is used to determine the cost thru computational cost of responding to query. Every field or resolver of the GraphQL schema is stipulated with a particular weight depending on its complexity to retrieve data, database manipulations, and anticipated processing time. These weights are summed up in the module to provide a cost estimate of the query. The system can identify a more precise response to query depth and the complexity of fields that is likely to be used to make additional predictions and avoid wasteful, overpriced queries.

3.1.4. Dynamic Rate Limiter

Dynamic Rate Limiter can be defined as the regulator of the amount of queries that a client can make to the system within a time duration. This module is not restrictive like in the traditional cases of rate limiting where only the number of requests are counted rather than considering the cost of the query, which is estimated. Among queries with a greater complexity, more request quota are used, whereas simpler queries require less resources. This dynamic still approach provides a balanced use of resources, eliminates wasted resource usage, and provides stability of the system in case of changing traffic situations.

3.1.5. Federated Gateway Controller

The Federated Gateway Controller handles communication between a federated architecture of different GraphQL services. In large scale applications, data can be split into many microservices and each has to deal with a specific section of the schema. The gateway attends to query routing, compiles responses of various services, and implements security and rates limitation measures at one of the central entry points. This module makes sure that the entire system is functioning efficiently without any variances in the security controls across the federated services.

3.2. Query Depth Calculation

Another relevant technique in the analysis of the graphical complexity of a GraphQL query is query depth calculation. GraphQL also allows the client to query data by any degree of nested fields, each degree being a relationship between the GraphQL schema objects. [12,13] Although it provides the advantages that the client can visit complex data structures with a single request, it may pose performance and security challenges when it becomes too deep. Thus, estimating query depth is useful in determining potentially costly/malicious queries that may be an early indication of that query being an expensive query prior to the query execution at the server. GraphQL query depth is computed through exploration of all potential paths of a query starting with the root field of that query to the deepest nested field. A path constitutes a list of fields that are linked between the point of query initiation and a leaf node in the query tree. As an example, we may have a query that begins with a user field followed by a request on posts relative to that user after which the comments by the respective posts are requested. The navigation would be user, posts, comments in this instance.

This path would be three since it has three tiers of the nested fields. The computation of the depth can be conceptualized as given below: the depth of a query Q is the size of the longest path in the query tree. Put differently, the system considers all possible paths in the query structure and draws out the one that contains the highest number of nested levels. The maximum value is the depth of the query in general. In order to carry out this calculation a query is firstly broken down into a tree format referred to as an Abstract Syntax Tree (AST). The tree items depict each field ordered by the client. It then costs the tree evaluating algorithms like the depth-first search to calculate the number of nested levels down each of the paths. The system is able to estimate query complexity by computing the longest path in the query tree. This depth value is compared to set security thresholds. In case of a depth that is beyond what is allowed, the system can deny the query or impose further restriction like harder rate limiting. GraphQL servers may mitigate resource exhaustion attacks by imposing depth limitations meaning that the queries are manageable and efficient to run.

3.3. Semantic Cost Estimation

A method of estimating semantic costs is used to determine the amount of computation needed to execute a GraphQL query. Contrary to the mere query counting methods, semantic cost estimation takes into consideration the complexity of each field and resolver that is being used in the query. [14,15] Because GraphQL provides the ability to request several nested fields in one query, one query can be much more intensive to process when it has largely the same size as it appears to be. Thus, the estimation of the semantic cost assists the system in knowing the cost of a query in advance. The semantic cost of a query involves computing summative cost of all the requested fields in the query and giving them a cost value. Every standpoint of the GraphQL schema is linked to a resolver that fetches or calculates the needed data. There are resolvers that can just give a value out of the memory, whereas other resolvers can do much more complicated tasks like a database query, API call, or data aggregation. In consequence of such differences, resolvency of each is attributed a particular weight, and this weight is proportional to its relative complexity. This weight gives an idea of how much power-consuming the resolver is supposed to become through its implementation. Simply put, the weight of a resolver is multiplied by the cost of the calculation and then the sum of the products obtained is each divided by the weight of each resolver in the query. Weight is the level of importance or complexity of the resolver whereas the cost of computation is the approximate processing cost to access the requested information. These two values are multiplied and the system ascertains the individual cost of carrying out that resolver. Once the cost of each of the fields used in the query has been computed, all of these values are added together by the system to get the total semantic cost of the query. Recently, one can use the example of multiple fields requested in one query, e.g. user information, posts, and comments, the corresponding weight, and cost of computation of the resolver will be in place. These values are multiplied by the system per resolver and added up to generate the total cost of the query. This last value is the sum of resource consumption that is expected to be used to process the request. The obtained semantic cost is subsequently contrasted with set radios that are set by the system administrator. In case the estimated cost is more than the acceptable limit, the query are rejected or throttled using rate limiting mechanism. The system is able to eliminate the likelihood of overloading of the server because of using expensive queries by means of semantic cost estimation, and thus, efficient utilization of resources.

3.4. Adaptive Rate Limiting

Adaptive rate limiting is a superior process utilized to regulate requests that a client may initiate to a system with regard to the computational complexity of the request. [16,17] Conventional techniques of rate limiting normally limit the count of requests per time interval, yet it does not take into account the resource-intensity of each request. In GraphQL systems though, the processing needs of different queries may vary very widely. A basic query can be asked to provide just a few fields, whereas a more involved query can carry out extensive nesting and multiple resolver. Owing to this inconsistency, adaptive rate limiting is intended to dynamically regulate the limit rate of allowable request rate, as per the approximate cost of each query. Adaptive rate limiting mechanism applies a formula to calculate the maximum permissible rate of requests by dividing the total processing capability of the system by the approximate cost of the requesting query. Simply put, allowed request rate ® is given as the threshold capacity of the system (T)/ cost of query which is denoted as Cost(Q).

System threshold capacity the highest safe level of computation that the system can manage during a specific time interval without compromising any of the performance or stability is known as system threshold capacity. The semantic cost estimation process, which is used to determine the cost of the query, estimates the complexity of query in terms of resolver weights and cost of computation. This calculation is done to make sure that, more complicated queries are charged at a reduced rate, and simple queries can be taken up at a higher number. As an example, when a query is computationally expensive, the system will limit the amount of requests that the client is permitted to issue so as to avoid server overload. Conversely, when the query cost is small, then the system might be able to support a larger request rate since the query will need few resources to execute. Dynamic rate limiting allows request limits to be adjusted dynamically according to request complexity, which offers more defense against resource exhaustion attacks and bad queries. It also guarantees equitable distribution of resources to various clients and assists to sustain the performance of the systems even in the times of high traffic. It can be especially helpful in the GraphQL-based systems where query structure and computation requirements differ dramatically across different requests.

4. Results and Discussion

4.1. Experimental Setup

The experimentation framework of this research was to test the efficiency of the proposed GraphQL security and rate-limiting framework in a simulated open banking network. The experiment aimed at studying the behavior of the system in dealing with complex queries and managing resources as well as supporting the system performance when several services are used over a federated architecture. To do so, a controlled testing environment was designed, which mimics the design of a modern open banking platform, with various financial services interacting via secure APIs. The environment consists of a GraphQL server, multiple backend microservices and a dataset with real banking transactions. The GraphQL gateway is the central node of the architecture that serves as a gateway of all incoming API requests. The gateway will be based on a GraphQL federation model, the federation that will enable numerous independent microservices to add to a single schema. This implies that customers need to make only a single GraphQL query to the gateway and the gateway will divide various sections of this query to the relevant services at the backend. The gateway also undertakes the implementation process of the suggested

modules of query parsing, depth analysis, semantic cost estimation and adaptive rate limiting. The back-end system is a collection of various microservices that are the embodiment of various banking functions. These services are the Account Service, Payment Service and the Transaction Service. Account Service is the account that deals with the user account details including account balance and account information. The payment service is the one that does the processing of payments and transfers of funds between accounts. The Financial activities that the Transaction Service records include deposits, withdrawals, and history of transactions. All services work separately but become interconnected using the GraphQL federation layer. A banking transaction data was employed to model the experimental setting and replicate actual banking operations. The data that is contained in this dataset comprises of customer account details, transaction records, payment activities, and account balances. The dataset is useful in creating real queries and workloads to test the system.

This experimental setup will allow studying the performance and security behavior of the proposed framework in the conditions typical of the real world open banking systems.

4.2. Performance Evaluation

The efficiency of the suggested Dynamic Banking Secure Rate Limiting (DBSRL) framework was tested and contrasted with the two other systems running without any protection and systems running with query depth limiting mechanism alone. The effectiveness of the framework was measured using several key performance measures such as the rate of attack detection, a reduction in the CPU usage, efficiency in query processing, reduction of false positives, and the service availability. The findings indicate that the suggested DBSRL model can enhance the security and performance of the system in a GraphQL-based open banking system to a considerable extent.

Table 1. Performance Evaluation

Metric	Without Protection	Depth Limit	Proposed DBSRL
Attack Detection Rate	52%	74%	94%
CPU Utilization Reduction	20%	48%	71%
Query Processing Efficiency	55%	68%	89%
False Positive Reduction	40%	60%	86%
Service Availability	62%	78%	93%

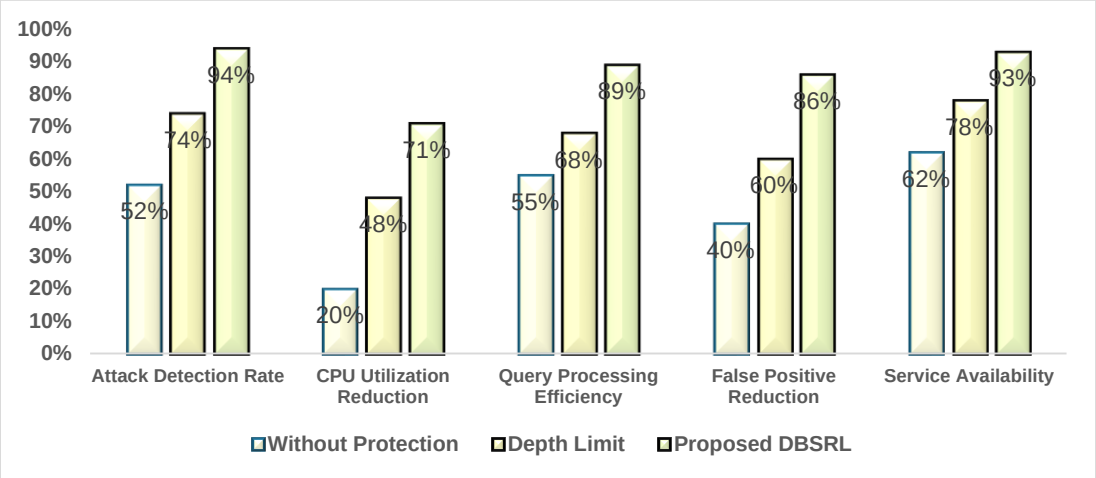

Figure 3. Performance Evaluation

4.2.1. Attack Detection Rate

Attack detection rate describes the capability of the system to detect and block malicious or excessively complicated queries that can lead to a performance degradation or denial-of-service assault. The detection rate was only 52 in systems that were not secured thus allowing a number of malicious queries to get through the system. The rate of detection increased to 74 percent when a depth limit mechanism was implemented since deeply nested queries were able to be detected and restricted. Nonetheless, the suggested DBSRL model had a detection rate of 94 per cent using a combination of query depth examination, semantic cost estimation, and adaptive rate restricting. Such a combined method enables the system to identify more complex and possibly dangerous queries.

4.2.2. CPU Utilization Reduction

Measures of CPU utilization reduction determine the efficiency of the system to reduce unnecessary processing load due to complex or malicious queries. The system suffered a cut in CPU utilization only to 20 percent in the absence of protection mechanisms since most expensive queries were still run. In the event depth limiting was enforced, CPU utilization reduction

went up to 48 since deeply nested queries were partially regulated. The proposed DBSRL framework had made significant improvement with 71 percent decrease in CPU usage. This is made possible by the fact that the system detects costly queries at an early stage and limits them prior to wasting a lot of computation resources.

4.2.3. Query Processing Efficiency

Query processing efficiency implies the degree to which the system can respond to legitimate queries with the entertainment of the performance. The efficiency was estimated at about 55 without any form of protection since complex queries usually slowed down the system. Depth limiting alone brought efficiency up to 68, as some costly queries were filtered. The suggested DBSRL model demonstrated 89 percent efficiency on the basis of the structure and cost of queries. This is to make sure that a query that is valid is executed without any hitches and the query that may have high resource consumption is limited or denied.

4.2.4. False Positive Reduction

False positives are those genuine queries that are detected as malicious or too complicated. False positive rate is also high and this may hurt the user experience and reliability of the system. False positive reduction was approximately 40 in non-advanced protection systems. With the application of depth limiting, it increased to 60 but still there were valid queries which were blocked by the heavy depth limits. Semantic cost estimation and adaptive rate limiting used in the proposed DBSRL framework greatly decreased false positives to 86% because they offer more precise information on query evaluation.

4.2.5. Service Availability

Service availability is used to gauge the capability of the system to be up and running even when it is in full traffic or attack modes. The availability of services without protection was just at 62% since resource-consuming queries would cause the server to crash. As depth was limited, the availability went up to 78% since the complex queries were limited. The proposed DBSRL framework had the rate of service availability of 93% through successful management of query complexity and request rate. This is to make sure that the system is always stable and available to the legitimate users even when there is high demand or even when attempts are made to attack it.

4.3. Discussion

The experimental findings indicate that, the discussed Dynamic Banking Secure Rate Limiting (DBSRL) framework leads to the significant enhancement of the security, efficiency, and reliability of open banking systems based on GraphQL. Basic API protection systems generally employ only basic request counting or fixed rate limiting policies, which fail to consider the different computational complexity of GraphQL queries. Consequently, these systems lack sophisticated protection features and risks of complex query attacks, deeply nested queries and resource exhaustion. The experimental test makes it evident that simple methods of protection are only partially useful in terms of defense against such threats. In the case of query depth limiting as the only form of control, the system is able to partially contain overly nested queries. This will enhance performance and some of the possible security risks, as observed in the enhanced attack detection rate and CPU consumption decrease. But depth limiting is not enough since a lot of GraphQL queries can seem shallow, but still consume a lot of computing power since there is a complex resolver operation or a large data retrieval operation. Thus, depth-based security schemes are not able to reflect the real computational cost of a query.

The suggested framework of the DBSRL system attempts to overcome these shortcomings by integrating a variety of protective schemes, such as query depth analysis, estimated semantic cost, as well as adaptive rate limiting. Within a system that analyses both the computational cost and the structural complexity of a particular query, it will be able to estimate the amount of processing power that the query is going to need. This allows the framework to identify costly queries in a more efficient manner and distributes resources in a more efficient way. As can be seen, this combined method can drastically enhance attack detection, decrease the CPU usage, and enhance query processing in general. The other significant benefit of the planned system is that it will minimize false positives. Conventional protection systems can be inaccurate and block any valid query, which affects user experience and system usability. The DBSRL framework offers better evaluation of queries by estimating semantic costs and controlling requests based on their priority so that the valid requests are handled without unwarranted restrictions. On the whole, the discussion emphasizes the fact that a combination of various security and performance control mechanisms is a more effective way of securing GraphQL API in open banking settings. The suggested framework is also set to increase the security of the system but also optimize the availability and performance of services in the event of a large workload.

5. Conclusion

The paper described a new Depth-Bounded Semantic Rate Limiting (DBSRL) model that can address the problem of the attacks on the complexity of the algorithm in federated GraphQL-based applications found in open banking environments. As the usage of GraphQL in contemporary API-based financial systems grows, another significant issue at hand is the security and efficiency of API communication. GraphQL enables highly flexible and nested data structures to be queried by clients, which is easier to use but creates potential security vulnerabilities. Attackers can use such flexibility to build deeply nested or

computationally expensive queries, which will use up excessive server resources and eventually may result in service degradation or even denial-of-service situations. To overcome this problem, the proposed framework will utilize a combination of structural query analysis and semantic cost estimation to dynamically control access to APIs and avoid this type of attack. The framework works by examining the structure and the computational complexity of the queries in GraphQL entered into it before they are executed. Structural analysis is conducted by query depth analysis that detects excessively nested queries that can lead to performance problems. Along with depth analysis, the system uses semantic cost estimation, in which each resolver and field of query is weighted by its computational complexity.

The framework sums up these weights to come up with the total cost of executing the query. The adaptive rate limiting mechanism then uses this cost value in dynamically regulating the amount of requests that a client can have. Queries of higher estimated costs are permitted at a lower rate and simpler queries can be executed more frequently. The strategy will guarantee the effective use of resources and stability of the system. The proposed structure was proved to be effective through the experimental analysis carried out in the simulated open banking environment. The findings indicated that major performance measures were improved considerably such as the rate of attack detection, reduction of CPU usage, efficiency in query processing, false positive, and service availability. The proposed DBSRL framework offers a more adequate protection mechanism than the traditional query depth limiting methods in that both query structure and computational complexity are taken into account. This enables the system to better differentiate between valid complex queries and malicious or resource-consuming queries. Additionally, the framework enhances system flexibility in federated GraphQL systems, whereby multiple microservices are used to add to a single schema. The system guarantees steady protection of distributed services by enforcing security and rate limiting on a gateway level, in addition to efficient execution of queries. This centralized control mechanism strengthens the security of APIs on the whole but does not impose a major impact on the legitimate user requests. This work can be further extended in the future by incorporating machine learning methodology to predict query cost and detect anomaly more precisely and automatically. Machine learning models would be able to study historical query patterns and detect abnormal behavior and dynamically determine security policy. Also, more sophisticated systems of detecting anomalies might be used to identify novel attack patterns against GraphQL APIs. Such improvements would also make GraphQL-based open banking systems more secure and resilient to the constantly evolving and more intricate digital financial ecosystems.

References

[1] Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures. University of California, Irvine.

[2] Hartig, O., & Pérez, J. (2018, April). Semantics and complexity of GraphQL. In Proceedings of the 2018 World Wide Web Conference (pp. 1155-1164).

[3] Xavier, L., Ferreira, F., Brito, R., & Valente, M. T. (2020, June). Beyond the code: Mining self-admitted technical debt in issue tracker systems. In Proceedings of the 17th international conference on mining software repositories (pp. 137-146).

[4] Hardt, D. (2012). The OAuth 2.0 authorization framework (No. rfc6749).

[5] Lodderstedt, T., McGloin, M., & Hunt, P. (2013). OAuth 2.0 threat model and security considerations (No. rfc6819).

[6] Al-Fares, M., Loukissas, A., & Vahdat, A. (2008). A scalable, commodity data center network architecture. ACM SIGCOMM computer communication review, 38(4), 63-74.

[7] Zachariadis, M. (2020). How “open” is the future of banking? Data sharing and open data frameworks in financial services. The Technological Revolution in Financial Services. How Banks, FinTechs, and Customers Win Together, 129-157.

[8] Premchand, A., & Choudhry, A. (2018, February). Open banking & APIs for transformation in banking. In 2018 international conference on communication, computing and internet of things (IC3IoT) (pp. 25-29). IEEE.

[9] Pappula, K. K., & Anasuri, S. (2021). API Composition at Scale: GraphQL Federation vs. REST Aggregation. International Journal of Emerging Trends in Computer Science and Information Technology, 2(2), 54-64.

[10] Haris, M., Farfar, K. E., Stocker, M., & Auer, S. (2021, November). Federating scholarly infrastructures with GraphQL. In International Conference on Asian Digital Libraries (pp. 308-324). Cham: Springer International Publishing.

[11] Brito, G., Mombach, T., & Valente, M. T. (2019, February). Migrating to GraphQL: A practical assessment. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 140-150). IEEE.

[12] Hantouti, H., Benamar, N., Taleb, T., & Laghrissi, A. (2018). Traffic steering for service function chaining. IEEE Communications Surveys & Tutorials, 21(1), 487-507.

[13] Stünkel, P., von Bargen, O., Rutle, A., & Lamo, Y. (2020). GraphQL Federation: A Model-Based Approach. J. Object Technol., 19(2), 18-1.

[14] Touronen, V. (2019). Microservice architecture patterns with GraphQL. University of Helsinki.

[15] Kellezi, D., Boegelund, C., & Meng, W. (2019, December). Towards secure open banking architecture: an evaluation with OWASP. In International Conference on Network and System Security (pp. 185-198). Cham: Springer International Publishing.

[16] Cha, A., Wittern, E., Baudart, G., Davis, J. C., Mandel, L., & Laredo, J. A. (2020, November). A principled approach to GraphQL query cost analysis. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 257-268).

[17] Desnitsky, V., Kotenko, I., & Zakoldaev, D. (2019). Evaluation of resource exhaustion attacks against wireless mobile devices. Electronics, 8(5), 500.
[18] Groza, B., & Minea, M. (2011, March). Formal modelling and automatic detection of resource exhaustion attacks. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (pp. 326-333).
[19] Mavroudeas, G., Baudart, G., Cha, A., Hirzel, M., Laredo, J. A., Magdon-Ismail, M., ... & Wittern, E. (2021, November). Learning GraphQL query cost. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 1146-1150). IEEE.
[20] Kua, J., Armitage, G., & Branch, P. (2017). A survey of rate adaptation techniques for dynamic adaptive streaming over HTTP. IEEE Communications Surveys & Tutorials, 19(3), 1842-1866.
[21] Chennareddy, R. K. (2020). Engineering Intelligence Systems Using Big Data and Cloud Architectures for Modern Data Intensive Applications. International Journal of AI, BigData, Computational and Management Studies, 1(2), 41-50.
[22] Chennareddy, R. K. (2021). Designing Data and Analytics Ecosystems for High Volume Transaction Processing Applications. International Journal of AI, BigData, Computational and Management Studies, 2(2), 95-106.