Performance Analysis on Multi-Party Computation "Online-Store" with Homomorphic CKKS encryption

Mohamad Falah Sutawindaya
School of Electrical Engineering and Informatics Bandung Institute of Technology
Bandung, Indonesia
falahsuta@gmail.com

Rinaldi Munir
School of Electrical Engineering and Informatics Bandung Institute of Technology
Bandung, Indonesia
rinaldi@informatika.org

***Abstract*—The problem of data breaches is increasingly widespread, online service providers often miss out, resulting in data service users being exposed. Online services can also be referred to as a cloud system called semi-trusted-cloud-source, homomorphic encryption is considered as one of the answers to provide solutions for exposed data as well as practicality in carrying out operations on the data. Various architectures of cloud systems can be implemented homomorphically, one of which is Multi-Party Computation (MPC) combined with homomorphic encryption that can provide internal data security for semi-trusted source services (various architectural models have their advantages and disadvantages). Seeing how big the impact of homomorphic implementation on hardware is also taken into account because it is related to costs in providing service providers, this study draws several conclusions about how much the additional cost is, which briefly requires one additional piece of hardware as a service worker for the needs of one cycle homomorphic computation. This paper also mentions how big the impact on storage requirements as one of the MPC-based modules. In the end, the implementation test can still provide an MPC system that runs well which involves computational homomorphism for its use with the scenario of storing transaction and balance data encrypted in the case study "online store". (*Abstract*)**

***Keywords*—Multi-Party Computation, Homomorphic, Web Service, Exposed Data, Data-Breach, Cloud-System, Semi-trusted-source, spike-testing, hardware-monitoring, Cryptography.**

I. INTRODUCTION

The adoption of cloud computing which has massive growth is not without reason, convenience and flexibility are the reasons for using cloud computing services instead of using traditional dedicated servers, which generally users do not need to perform maintenance on the related server because this has been done by service providers only need to configure on the usage and computing needs he wants (Sangeeta, 2012). Cloud service providers also generally provide convenience in managing the resources needed so that the system can serve users well. The flexibility of the use of cloud services is also a matter of concern because the implementation of the architecture in the program that the user is trying to install is not necessarily safe, so this is a problem in itself because the user trusts the data.

User data itself has the greatest risk during the idle stage or when user data is stored on cloud computing service resources (Vijay, 2014). This is an event called data-breach, i.e. a stolen database event that exposes millions of user data (Hicham, 2019). The use of a different architecture such as blockchain requires high-cost technical investments, massive changes to the way centralized systems generally work. The easiest and cheapest way from technical costs but still provides more benefits to data security is to implement a layer or encryption layer on the data stored in the hexagon and stored (Shivani, 2021). The method of applying encryption to data is as important as data security because if it is not appropriate, it will cause swelling of computing resources and reduce the effectiveness of cloud computing services.

Homomorphic encryption with the initial idea of using it is intended to be able to privatize data on external or external resources in a cooperative environment in commercial cloud computing services, user data such as health records, addresses, credit information or other personal information can still maintain its confidential condition even if a data breach occurs. and system errors. For this reason, optimizing the use of encryption algorithms in programs running on cloud computing services is one step to make the cloud computing system more secure because it can overcome the occurrence of data theft scenarios (Raymond, 2015).

One of the technical schemes that are currently known, one that offers good speed and is claimed to be usable is the CKKS scheme (Cheon-Kim-Kim-Song), this scheme offers arithmetic operations that approximate numbers that can be used on the original numbers which makes it an advantage. To see how far the use of CKKS on hardware in this paper is tested on a medium-scale traffic to see the response of a system that uses CKKS where the use of this algorithm promises to be one of the appropriate solutions for Multi-Party Computation (MPC).

To start, in this paper we present the implementation of Multi-Party Computation on the "Online Store Scenario" with a series of tests on hardware to see the impact of using CKKS homomorphic encryption on the system.

II. CKKS ALGORITHM

The CKKS algorithm is one of the newest homomorphic algorithms adapted from the SEAL library which is fully-homomorphic-encryption (FHE), along with BFV. CKKS and BFV are available in the Golang language which is available in

a library called latigo which is a conversion from SEAL which is a library written in C++. Writing on Golang brings new possibilities regarding homomorphic implementations to the web services. The benchmarks on the official service show very similar performance between these two languages. CKKS allows operations up to complex numbers, while BFV only allows operations on integers. The advantages of using CKKS can be adopted in more cases in web services as exemplified in the development of this paper solution.

A. Key Generation

To see the equivalent of the sequence of mathematical operations with writing programs, let's look at the following steps:

To create a secret key and a public key, let's look at the following equation

$sk = s$

$pk = (a, b = -a \,.\, s + e)$,

where $a,s,e \in R_q$, R_q cycle polynomial ring, and $sk \in R_q$, while $pk \in R_q^2$ is a vector polynomial with two dimensions (public-key).

B. Encoding and Decoding

The encoding process converts plaintext (in the form of complex vectors or float data types for example) into members of a polynomial ring, which is defined as follows, for example $z \in C^{n/2}$ is a composition vector with slots or a complex number space, and is a scale on the plaintext associated with precision. The encoding algorithm process is:

$$\mu \leftarrow \text{Ecd}(\Delta, z)$$

With $\mu \in R = Z(x) / x^n + 1$, or a plaintext that belongs to a polynomial ring. For the decoding process that will change the plaintext that is not a member of the polynomial ring, the decoding algorithm process is as follows:

$z' \leftarrow \text{Dcd}(\Delta, \mu)$

where $z' \in C^{n/2}$ is a complex vector or float data type of dimension n/2, provided that $z' \approx z$.

C. Encryption and Decryption

For the encryption process is the process before the operation is carried out on the ciphertext results, by looking at the following equation is the mathematical process of performing encryption. Given a plaintext, namely which has the property $\mu \in R$, and $pk = (-a \cdot s + e, a) \in R_q^2$. The encryption algorithm is as follows:

$$c \leftarrow \text{Enc}(pk, \mu),\ c = (\mu - a \cdot s + e, a) = (c_0, c_1) \in R_q^2$$

where c is the ciphertext which is $c \in R_q^2$ which means that each component is a polynomial ring. While the reverse process is decryption, the way the algorithm works is as follows:

$$\mu' \leftarrow \text{Dec}(s, c),\ \mu' = c0 + c1 \cdot s$$

$$\mu' = \mu - a \cdot s + e + a \cdot s = \mu + e \approx \mu$$

With the same properties $c \in R_q^2$, and $sk = s$, and also $\mu \in R$ which is the plaintext form.

D. Ciphertext-Plaintext Addition

With $\mu \in R$ for the plaintext and $c = (c_0, c_1) \in R_q^2$ for the ciphertext, the addition of ciphertext c and plaintext , follows the following calculation:

$$c_{add} = c + \mu = (c_0 + \mu, c_1)$$

with $sk = s \in R$ and the decryption process $\mu' \leftarrow \text{Dec}(sk, c)$ which is the decryption of ciphertext c, it can be verified that c_{add} is a close approximation to $\mu' + \mu$ which is a mathematical form of equation with the addition of each ciphertext and the plaintext is expressed by the following equation:

$$\text{Dec}(sk, c_{add}) = c_{add},0 + c_{add},1 \cdot s = c_0 + \mu + c_1 \cdot s = \mu + \mu' - a \cdot s + a \cdot s + e \approx \mu + \mu'.$$

E. Ciphertext-Plaintext Multiplication

Meanwhile, in the process of multiplication with initiation, with $\mu \in R$ for the plaintext and $c = (c_0, c_1) \in R_q^2$ for the ciphertext, the multiplication of ciphertext c and plaintext μ, follows the following calculation:

$$c_{mult} = (\mu' \cdot c_0, \mu' \cdot c_1)$$

then it can be verified that cmult is a close approximation to $+\mu$ which is a mathematical form of an equation with the product of each ciphertext and plaintext expressed in the following equation:

$$\text{Dec}(sk, c_{mult}) = \mu' \cdot c_0 + \mu' \cdot c_1 \cdot s = \mu' \cdot (c0 + c1 \cdot s) = \mu' \cdot (\mu - a \cdot s + e + a \cdot s) = \mu' \cdot \mu + \mu' \cdot e \approx \mu' \cdot \mu$$

F. Ciphertext-Ciphertext Addition

Meanwhile, in the process of adding ciphertext c with ciphertext c', for example $c = (c_0, c_1)$, $c' = (c'_0, c'_1) \in R_q^2$ for two ciphertexts to be added, the addition is defined as follows:

$$c_{add} = c + c' = (c_0 + c'_0, c_1 + c'_1)$$

With $sk = s \in R$ and $\mu \leftarrow \text{Dec}(sk, c)$, $\mu' \leftarrow \text{Dec}(sk, c')$ being the decryption of c, c' it can be verified that c_{add} is a close approximation to $+\mu$ which is of the form Mathematically the equation with the addition of two ciphertexts is stated by the following equation:

$$\text{Dec}(sk, c_{add}) = c_{add},0 + c_{add},1 \cdot s = c0 + c'0 + (c1 + c'1) \cdot s = \mu + \mu' + 2e \approx \mu + \mu'$$

G. Ciphertext-Ciphertext Multiplication

Meanwhile, in the process of multiplying ciphertext c with ciphertext c', for example $c = (c_0, c_1)$, $c' = (c'_0, c'_1) \in R_q^2$ for two ciphertexts to be added, the multiplication is defined as follows

$$c_{mult} = (c_0 \cdot c'_0,\ c_0 \cdot c'_1 + c'_0 \cdot c_1,\ c_1 \cdot c'_1) = (d_0, d_1, d_2)$$

With sk = s ∈ *R*, it can be verified that c_{mult} is a close approximation to the decryption process of the multiplication of two ciphertexts stated as follows:

$$Dec(sk,c) \cdot Dec(sk,c') = (c_0 + c_1 \cdot s) \cdot (c'_0 + c'_1 \cdot s) =$$

$$c_0 \cdot c'_0 + (c_0 \cdot c'_1 + c'_0 + c_1) \cdot s + (c_1 + c'_1) \cdot s^2 = d_0 + d_1 \cdot s + d^2 \cdot s^2$$

As the added multiplying process, it's best to include linearization primitive to reduce the size.

III. MULTY-PARTY HOMOMORPHIC ENCRYPTION PROGRAMS

To be able to show the performance of CKKS on medium-scale traffic, we created several inspired by case studies of an online store, we created several scenarios that will take advantage of the MPC scenario that uses the arithmetic operation feature that utilizes the CKKS feature, namely multiplication on discounts, and adding balances. users, and privatize purchase transaction data. Full program implementation source code will be added in the conclusion for the GitHub link.

A. Activate Balance (Balance Module)

Here is an "Activate Balance" module flow architecture design which is used to activate a user's balance if the user has never activated a balance before. The following is the architectural design.

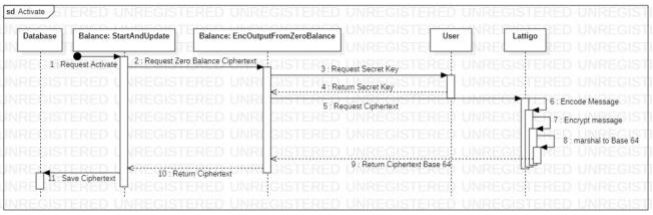

Fig. 1. Application Module (Balance)

B. Adding/Subtract Balance (Balance Module)

here is a "Top up Balance " module flow architecture design that is used to fill the user's balance when the user has activated the previous balance. The following is the architectural design.

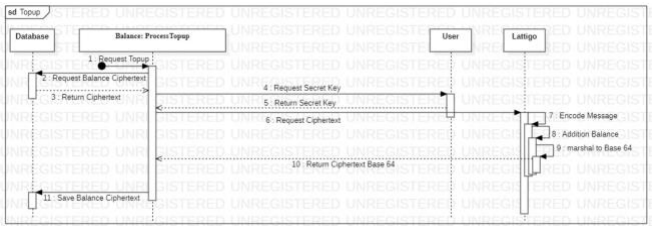

Fig. 2. Application Module (Balance)

C. Check Balance (Balance Module)

There is an architectural design for the "Check Balance" module flow that is used to decrypt the user balance ciphertext for display. The following is the architectural design.

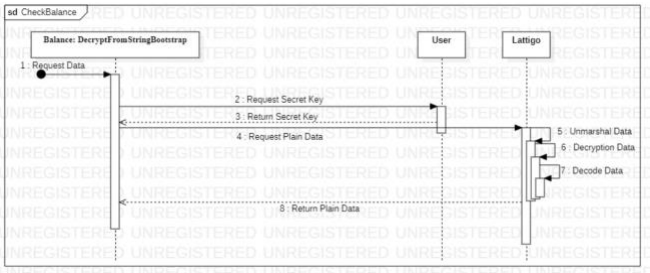

Fig. 3. Application Module (Balance)

D. Discounts on Item Purchase (Transacts Module)

Here is a flow architecture design for the "Create Transact" module which is used to perform product transactions using the user's balance if the user has previously activated the balance. The following is the architectural design.

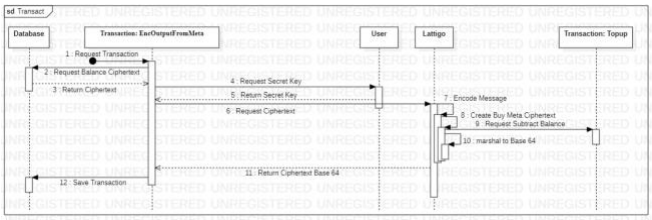

Fig. 4. Application Module (Transacts)

E. Private purchase Transaction data (Transacts Module)

Here is a "Get Private Transact" module flow architecture design that is used to get product transactions that have been done before. The following is the architectural design.

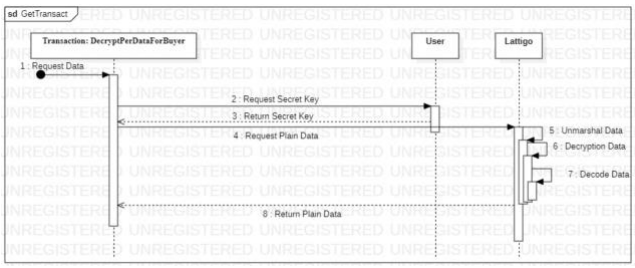

Fig. 5. Application Module (Transacts)

IV. PERFORMANCE TESTING

To see how this CKKS encryption affects the hardware, the following series of tests are carried out. The test starts from the lowest scale, which is to only test the speed of the program, then see the impact on programs that perform many operations at once and at the same time see the impact on hardware with monitoring tools.

A. *Program Execution Overhead Testing*

To be able to measure how much homomorphic computing penalizes the system (how much hardware resources are used) to see how fast or slow the execution of this homomorphic-based program takes up a portion of resources can answer the question of when this program takes a long enough pause to executed and overloaded the system.

With the system built, it is found that the required execution speed along with each degree value for the library parameters used and the scenario of calculating the price after the final discount (involves adding and multiplying the homomorphic ciphertext for the operation).

TABLE I. STANDARD CKKS OPERATION BENCHMARK

Operation	**d = 2^4**	**d = 2^5**	**d = 2^6**	**d = 2^7**	**d = 2^8**
Encode	19	17	30	44	90
Encrypt	51	76	137	249	914
Multiplication	10	8	10	12	21
Add	6	6	6	7	9
Decrypt	2	7	5	12	27
Decode	36	35	42	44	85
Overhead per Operation (in µs)	124	149	230	368	1146

The first test regarding the amount of delay that appears when adding computational homomorphic, the assumption of this test is obtained with a margin of error of +-20, in units of microseconds. This calculation or test assumption is based on the time required to execute homomorphic calculations or execution with external factors other than the program, for example delays or pings from connections to retrieve data from a network (e.g. storage database connections or service access connections). The last column describes the overhead, i.e. the time required to execute a full set of computational homomorphic whose value is obtained by adding up the values in the previous column. Other implications of using a higher degree apart from increased execution delays arise from demand storage or required storage.

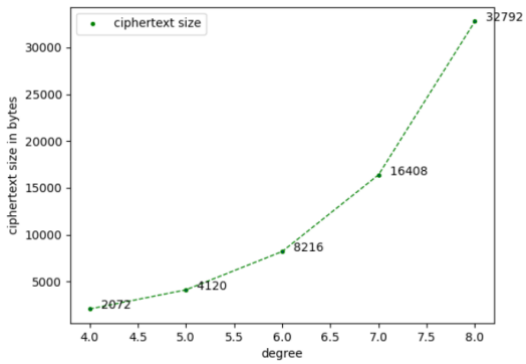

Fig. 6. Storage CKKS Encryption Requirement

Shows the various degrees of measurement and their effect on the size of the ciphertext, this is related to the calculation of the storage requirements of the system being built. It can be seen that from the table before, the size is magnified twice to a magnification of one degree, the graph of the relationship is linear even though it looks like an exponential because it uses a scale of 0.5 on the degree.

B. *Testing on Many Service Users*

System testing is also carried out with varying loads, which is not just a single-execution program to test how the system reacts at larger load scales. To see the performance penalty that occurs, the test is carried out on 2 different variables, computational homomorphic and computational without cryptography. The test scenario will use the "k6 load tester" library and to be able to see the stress-test the test will use the configuration that will be shown below with the output of the completed computational quantities. The scenario is tested on an API endpoint utilizing a full circuit homomorphic computationally (when using).

TABLE 2. SPIKE TESTING

Operation	**Finished Operation (3s)**	**Average Request Per Second (RPS)**
Without Homomorphic	2759	920
Homomorphic, d = 2^4	1157	385
Homomorphic, d = 2^5	839	280
Homomorphic, d = 2^6	592	197
Homomorphic, d = 2^7	325	108
Homomorphic, d = 2^8	164	54

Testing in Table 2 aims to see how the system reacts to computing for a larger load scale, the test scenario is carried out with a computational time cutoff (3s), after the computational limit will no longer accept computational requests. Testing is done by spawning as many requests as can be completed within the cutoff period. The test compares the system without homomorphic computations and with homomorphic computations to see the performance penalty given, it appears that the completed computations are reduced by 50%-75% with the lowest degree value for the same time.

C. *Hardware Performance Metrics in Graphs*

The scenario being tested is the scenario same as before, namely spike-testing by multiplying as many operations as possible that can be completed in 3 seconds. The following is a display of the impact that occurs on the hardware during the spike-testing.

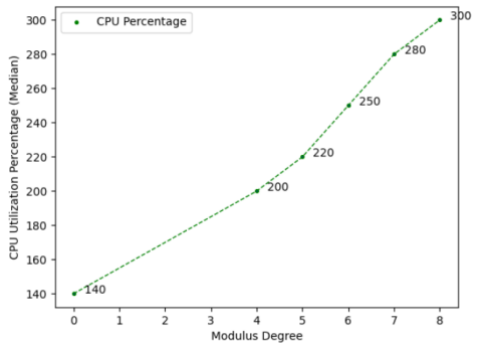

Fig. 7. CPU Utilization Comparison By Modulus Degree

For CPU utilization, it can be seen that for each increase in degree, it increases by 25-40% for an increase in utilization. As a graphic explanation, the degree of modulus with a value of 0 is computational without homomorphism. The trend of increasing CPU utilization is due to the increasing weight or number of ciphertexts that must be computed homomorphically.

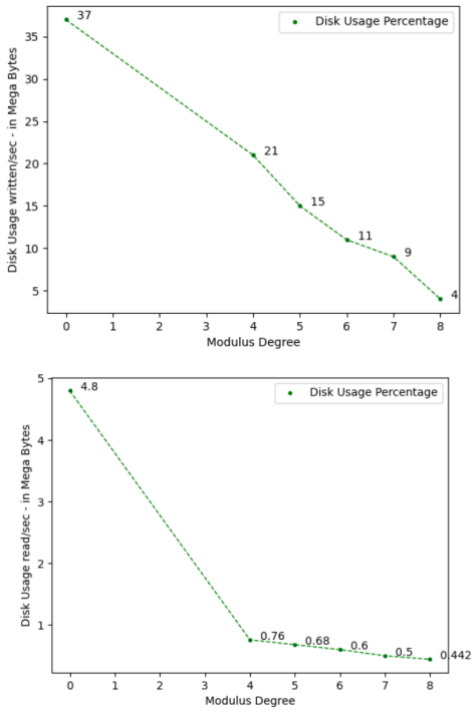

Fig. 8. Read/Write Rates Comparison By Modulus Degree

Figure 8 is linear with the Table 2, which decreases with increasing degrees of modulus, this is because fewer operations are completed so that data written or read on disk experiences a downward trend.

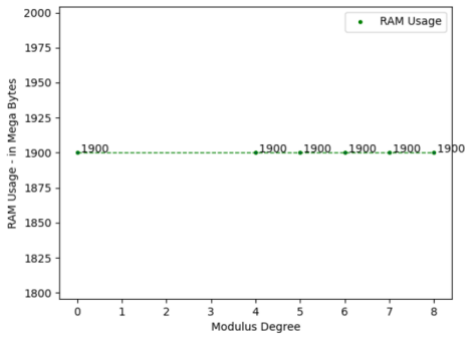

Fig. 9. Memory (RAM) Usage Comparison By Modulus Degree

Meanwhile, RAM usage tends to have no significant changes or spikes in resource usage, consistent with Figure IV-32, this can happen because the running program has a fixed-allocation so that memory usage remains constant despite the difference in load due to spike-testing. For this series of tests, it can be concluded at a glance that the program is CPU-bound, which means that homomorphic usage is dominated by more dominant CPU usage. Analysis and mitigation that can be used for system adoption is explained further in subsection IV.9, namely the analysis opinion regarding the test results.

V. CONCLUSION

In this paper we have tried to build a solution for an MPC "online store" application with CKKS using Golang as the server and JavaScript as the client, and a series of tests are shown to show the hardware specifications required. We found that the CPU usage increased by 60% for the lowest degree, and the encrypted data storage requirement required at least about 400-500% of the normal requirement at the lowest degree, as a comparison normal database column required about 20 bytes - 400 bytes with the lowest degree usage required at least more than 2 kilo bytes, regarding CPU usage we recommend adding at least one worker service to provide the computational needs for homomorphic computational overhead as it easier to add more vertical scaling solution than changing program structure for specific hardware requirements.

REFERENCES

[1] A. Carey, “On the Explanation and Implementation of Three Open-Source Fully Homomorphic Encryption Libraries,” pp. 7-9, 2020.

[2] J. Vacca, Cloud Computing Security Foundations and Challenges, CRC Press, 2020.

[3] Hicham, “Digging Deeper into Data Breaches: An Exploratory Data Analysis of Hacking Breaches Over Time,” Science Direct, pp. 2-4, 2019.

[4] Sangeeta, “Exploring the impact of Cloud Computing adoption on organizational flexibility: A client perspective,” IEEE, 2012.

[5] Vijay, “A Survey of Cryptographic Approaches to Securing Big-Data Analytics in the Cloud,” IEEE, pp. 1-6, 2014.

[6] Shivani, “What Is The Standard Cost Of Building A Blockchain Application?,” 2021. [Online]. Available: https://www.cisin.com/coffee-break/technology/what-is-the-standard-cost-of-building-a-blockchain-application.html.

[7] R. K. &. Raymond, The Cloud Security Ecosystem, Elsevier, 2015.

[8] B. &. Rogaway, Introduction to Modern Cryptography, UC Davis, 2015.

[9] D. &. Helmut, Introduction to Cryptography, New York: Springer, 2015.

[10] Encryption Consulting, “What is Cryptography in security? : Asymmetric, Symmetric, and Hashing,” [Online]. Available: https://www.encryptionconsulting.com/education-center/what-is-cryptography/.

[11] J. Kun, “Why Theoretical Computer Scientists Aren’t Worried About Privacy,” 2013. [Online]. Available: https://jeremykun.com/2013/06/10/why-theoretical-computer-scientists-arent-worried-about-privacy/.

[12] Bolboceanu, “a Toy Implementation in Python,” 2016. [Online]. Available: https://bit-ml.github.io/blog/post/homomorphic-encryption-toy-implementation-in-python/.

[13] S. Johnston, “Creative Commons,” 2013. [Online]. Available: https://creativecommons.org/licenses/by-sa/3.0.

[14] C. Wong, HTTP Pocket Reference, O'Reilly, 2000.

[15] D. Gourley, “HTTP: The Definitive Guide,” 2021. [Online]. Available: https://www.oreilly.com/library/view/http-the-definitive/1565925092/ch04s05.html.

[16] D. Mills, “Mozilla MDN Developer,” 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

[17] D. &. Anjo, “Automatically Securing Linux Application Containers in Untrusted Clouds,” 2019. [Online]. Available: https://slidetodoc.com/automatically-securing-linux-application-containers-in-untrusted-clouds/.

[18] I. I. Wuri, “Homomorphic Encryption versus RSA: Cloud Security Performance Analysis,” pp. 2-5, 2021.

[19] W. F. Infall, “Cloud Security Implementation using Homomorphic Encryption,” pp. 2-5, 2020.

[20] Ahmed, “A Verifiable Fully Homomorphic Encryption Scheme for Cloud Computing Security,” MDPI Technologies, pp. 1-15, 2019.

[21] A. Vázquez, “Study and Applications of Homomorphic Encryption Algorithms to Privacy Preserving SVM Inference for a Bank Fraud Detection Context,” Universidade de Vigo, pp. 23-31, 2020.

[22] L. Norris, “Analysis of Partially and Fully Homomorphic Encryption,” pp. 2-5, 2013.

[23] “HTTP request methods – REST API verbs,” [Online]. Available: https://nlogn.in/http-request-methods-rest-api-verbs/.

[24] “HTTP request methods,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods.

[25] S. Wang, “Big Data Privacy in Biomedical Research,” IEEE TRANSACTIONS ON BIG DATA, pp. 1-15, 2016.