International Journal of Innovative
Computing, Information and Control
Volume **18**, Number **6**, December **2022**

 ISSN 1349-4198
pp. 1701–1717

POST-QUANTUM HYBRID ENCRYPTION SCHEME FOR BLOCKCHAIN APPLICATION

Kevin Hendy and Arya Wicaksana*

Department of Informatics
Universitas Multimedia Nusantara
Jl. Scientia Boulevard, Tangerang, Banten 15810, Indonesia
kevin.hendy@student.umn.ac.id; *Corresponding author: arya.wicaksana@umn.ac.id

Received April 2022; revised July 2022

Abstract. *The blockchain has gained more adoption and attention due to its secure, transparent, and decentralized characteristics. The discovery of quantum computers and algorithms exposed a threat to the established security standard, including the blockchain. This study aims to implement Kyber, a cryptographic primitive included in Cryptographic Suite for Algebraic Lattices (CRYSTALS), along with AES-256 symmetric encryption algorithm in a hybrid encryption scheme for a blockchain application. The test and evaluation results using Hyperledger Caliper show that the key size 768 on CRYSTALS-Kyber provides the best performance due to stable encryption and decryption time compared to key sizes 512 and 1024. The peak throughput is 12.16 transactions per second for two nodes and 20.48 for four, and the latency is 3.396s for two workers and 3.544s for four. This approach adds Kyber as an extra module against quantum attacks. This addition reduces the throughput of Oricon by 57.03% for two workers and 52.15% for four. This study finds that post-quantum cryptography could be added to existing blockchain applications with minimum to no modifications to the blockchain applications and shows the performance trade-off with adding a post-quantum security module.*
Keywords: Blockchain, CRYSTALS-Kyber, Hybrid encryption, Post-quantum

1. **Introduction.** The establishment of quantum computing and algorithms poses threats to current security standards, such as the Rivest-Shamir-Adleman (RSA) cryptography that has been the industrial-grade standard for decades [1-5]. Shor's quantum factoring algorithm can find the factors of the RSA, exposing the cryptography in less time than classical computers [6-8]. Theoretically, a quantum computer can crack RSA-2048 encryption in 10 seconds if a stable 4,099 qubits quantum computer is found [9] and in 8 hours using 20 million noisy qubits [10]. A quantum-proof security standard is essential to prepare for the upcoming quantum computing era. Most computer applications today, including blockchain technology [11-13], are exposed to the threats presented by the post-quantum era [14].

The Oricon anti-counterfeiting application is a blockchain-based application that features physical goods identification and verification [15]. This Oricon product authentication application uses a blockchain platform created by IBM and the Linux Foundation: Hyperledger Fabric [16]. Hyperledger Fabric uses the asymmetric cryptographic algorithm Rivest-Shamir-Adleman (RSA) or the asymmetric cryptographic algorithm Elliptic Curve Digital Signature Algorithm (ECDSA) for the digital signatures, where this is used to sign transactions. Both RSA and ECDSA are vulnerable to exploitation in the post-quantum era. This issue has become a concern to the National Institute of Standards and Technology (NIST), which in 2020 gathered up a call for proposals for post-quantum

DOI: 10.24507/ijicic.18.06.1701

cryptosystems. The aim is to select several cryptosystems candidates that can be accepted as post-quantum cryptography standards [17]. One of the finalists is CRYSTALS-Kyber [18], an asymmetric public-key encryption algorithm designed with three key size options: Kyber512, Kyber768, and Kyber1024 [19]. However, asymmetric cryptography has a disadvantage for blockchain applications due to its high cost and slow speed compared to symmetric cryptography [20]. Therefore, a hybrid encryption scheme is proposed in this study for blockchain application: Oricon.

Related work on this problem is carried out by the Quantum-Resistant Ledger (QRL) [21]. The work focused on developing a brand new blockchain system that is post-quantum secure and employs post-quantum computing technologies in its design for absolute security, audited by red4sec and x41 D-sec. Other work in [14] works on the concept of quantum resistance in blockchain networks by focusing on four areas: digital signatures, communication over the Internet, block mining, and hash functions. Another work in [22] aims to study the current post-quantum cryptography technologies. It is concluded from that study that most of the post-quantum cryptosystems were being analyzed by the NIST at the time of that study. In addition, that study concludes that choosing a blockchain post-quantum cryptosystem is not straightforward due to performance considerations, i.e., memory and speed. Based on the related works, this study intends to provide an approach feasible for many blockchain applications requiring minimum to no modifications to the blockchain applications, including the distributed ledger technologies, and consideration of the performance changes.

The main contribution of this paper is the hybrid encryption scheme for blockchain applications to withstand quantum attacks. The post-quantum cryptography CRYSTALS-Kyber is used in addition to the AES-256, and this hybrid encryption scheme is implemented for the Oricon blockchain application as proof of concept. This study aims to develop a security measure for blockchain applications to face the post-quantum era with minimum modifications required on the blockchain applications. The implementation of the hybrid encryption scheme is measured by the performance in terms of successful transactions, failed transactions, latency, and throughput generated, as well as the time required to perform key generation, encryption, and decryption of the implementation of the post-quantum cryptography CRYSTALS-Kyber algorithm with different key sizes.

The rest of this paper is divided into four sections. Section 2 presents literature related to this work. Section 3 presents the methods, and Section 4 presents the results and analysis. Finally, Section 5 discusses and concludes the work conferred in this paper.

2. **Preliminaries.**

2.1. **CRYSTALS-Kyber.**

CRYSTALS-Kyber is one of two cryptographic algorithms proposed by the Cryptographic Suite for Algebraic Lattices (CRYSTALS) at the 2017 NIST post-quantum standardization effort. CRYSTALS-Kyber entered as a finalist for the public-key encryption and key-establishment algorithm in the third round of the call for proposals conducted by NIST. CRYSTALS-Kyber is a Key-Encapsulation Mechanism (KEM) resistant to IND-CCA2 (Indistinguishability under adaptive Chosen Ciphertext Attack) with a security base hardness of solving the learning-with-errors problem in module lattices. The CRYSTALS-Kyber construction consists of two parts, namely KYBER.CPAPKE and KYBER.CCAKEM. KYBER.CPAPKE is an asymmetric-encryption scheme that is resistant to IND-CPA while KYBER.CCAKEM is a key-encapsulation mechanism scheme resistant to IND-CCA2, the most robust security notion for a public-key encryption scheme [23].

The Key-Encapsulation Mechanism (KEM) aims to encapsulate randomly selected bit-strings (keys). The output generated from KEM is a randomly selected key and the result of the key encapsulation. This key is reused to encapsulate data with an asymmetric encryption scheme. This data encapsulation is known as the Data Encapsulation Mechanism (DEM). This scheme is also known as hybrid encryption or the KEM-DEM. A key encapsulation mechanism consists of three algorithms, that is [24]

1) KeyGeneration produces a pair of public and private keys.
2) Encapsulation accepts a public key as an input, and then produces a random key and a ciphertext as the output.
3) Decapsulation accepts a ciphertext and a private key and produces a key encapsulated in ciphertext as the output.

2.2. **Oricon.** Oricon is a semi-decentralized application (DApp) utilizing blockchain to store product information from its company. This DApp aims to authenticate the products registered in it. Oricon is built using Hyperledger Fabric blockchain, and several functions can be run on the Oricon application [15].

1) Read Product is a function used to view product details desired by the user.
2) Update Product is a function that changes existing product data in the database, and product checking is done first before changing the product data.
3) Delete Product is a function to delete existing product data in the database.
4) Product History is a function that checks the flow of product data changes in the blockchain.
5) ERC20 token is a function implemented using the ERC20 token fungible standard. This ERC20 includes functionalities such as viewing the token name, token symbol, decimal used by the token, token supply amount, and user balance. This function can also transfer tokens from one user to another. The token transfer process requires the user's approval who receives the token. Users who receive tokens can determine the number of tokens to be received from other users. If the token supply is running low, this ERC20 function can add to the total supply of all tokens.

2.3. **Hyperledger Caliper.** Hyperledger Caliper is a tool supported by Hyperledger Fabric to perform benchmarking processes so that the performance of the implemented blockchain can be known. Performance measurement on Hyperledger Caliper requires several use cases [25]. In this research, the Hyperledger Caliper is used to measure performance from four aspects from [15].

1) Successful transaction is the number of valid transactions entered into the ledger. A transaction can be said to be valid if it passes the validation stage of the blockchain.
2) Failed transaction is the number of transactions that failed at the blockchain's endorsement and validation stage. Transactions that fail at the validation stage are entered into the ledger with invalid transaction status, and these transactions are excluded from the world state.
3) Latency is the time it takes for one transaction to be successfully stored in the ledger.
4) Throughput is the number of valid transactions successfully entered into the ledger within a specific period.

The results obtained from measuring performance in terms of successful transactions, failed transactions, latency, and throughput from the research conducted in [15] compare performance before and after the implementation of CRYSTALS-Kyber on smart contracts. Use cases used are that BatchTimeout is set to two seconds, MaxMessageCount is set to twelve messages, and the number of workers is set to two and four. This setting follows the configuration of the previous study on Oricon to provide a direct comparison

Table 1. Hyperledger Caliper 2 workers experiment results

Name	Successful transactions	Failed transactions	Latency (s)	Throughput (TPS)
createProduct	21	86	1.4300	0.6000
readProduct	1,186	1,451	0.0400	39.5000
readProduct History	3,021	0	0.0200	100.7000
updateProduct	14	133	1.9900	0.4000
deleteProduct	11	89	2.2700	0.3000
Average	850.6000	351.8000	1.1500	28.3000

Table 2. Hyperledger Caliper 4 workers experiment results

Name	Successful transactions	Failed transactions	Latency (s)	Throughput (TPS)
createProduct	3	33	11.6700	0.1000
readProduct	1,628	24	0.0400	54.3000
readProduct History	4,780	0	0.0400	159.2000
updateProduct	7	60	3.7400	0.2000
deleteProduct	10	90	4.6800	0.2000
Average	1,285.6000	41.4000	4.0340	42.8000

between the additional post-quantum security module and the original. The results of the two experiments can be seen in Tables 1 and 2 [15].

3. **Methods.**

3.1. **Requirement analysis.** Oricon's smart contract is made of Hyperledger Fabric version 1.4.8 using Fabric SDK Node.js 1.4.0. The front end of the Oricon is made using Ionic v6.12.0 and Angular v10.0.14 on Windows 10 64-bit Intel Core i7-8750H using 2.20 GHz CPU, 20 Gigabytes RAM, and 931 Gigabytes Hard Disk Drive. While the frontend is made on Windows, the smart contract is made on Linux Ubuntu Virtual Machine 20.4 64-bit using 2.20 GHz CPU and 10 Gigabytes RAM.

3.2. **Design.** The KeyGeneration function is executed only once when the smart contract is first executed. After the KeyGeneration function is executed, the client can use the Oricon application to process transactions such as retrieving, adding, or changing data. The Decryption function is called when the client wants to retrieve transaction data, and the Encryption function is called when the client wants to add or change data. These functions are implemented on the smart contract of the Oricon product authentication application using Node.Js SDK. Figure 1 shows the main flowchart showing the system's flow in outline.

A submenu is created in the Oricon product authentication application. The user can use the submenu key to change the key size of the CRYSTALS-Kyber post-quantum cryptography algorithm. This key size is used to determine the key size of the public key used during the encapsulation process of the symmetric key into its ciphertext and the private key used during the decapsulation process of the ciphertext into its symmetric key.

The key size selected by the user is stored in local storage. When the user sends a request to create or update a product to the server, this key size is retrieved from local

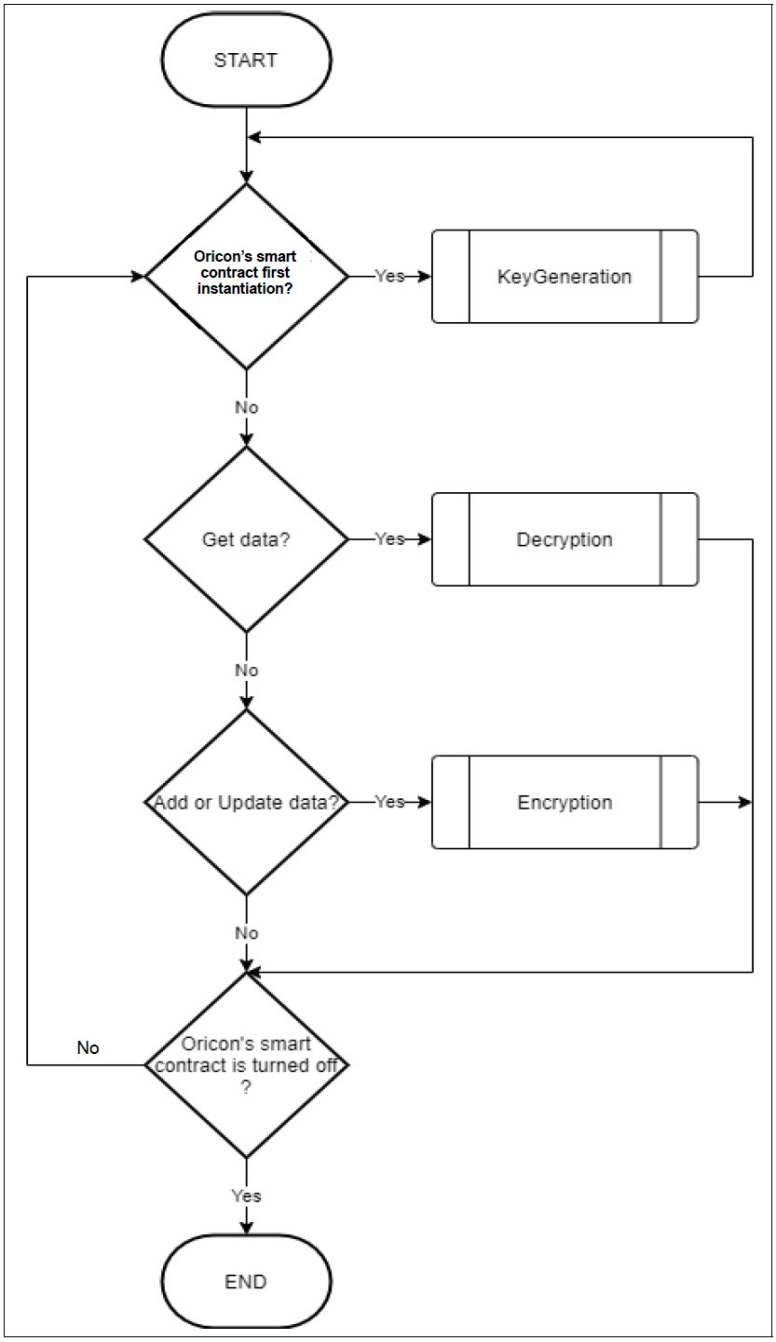

FIGURE 1. Main flowchart

storage and included in the JSON body. The key size is added to the URL parameter when the user sends a read product request.

3.3. **Implementation.** Hybrid encryption uses two encryption algorithms: the symmetric encryption algorithm AES-256 and the asymmetric encryption algorithm CRYSTALS-Kyber. The AES-256 algorithm is used to perform data encapsulation, while the CRYSTALS-Kyber algorithm is used to perform key encapsulation. In this study, encryption refers to data encapsulation, while encapsulation refers to key encapsulation to facilitate understanding. The declaration of configuration variables needs to be done before the encapsulation and decapsulation functions can be used. Figure 3 shows a snippet of the hybrid encryption algorithm configuration code.

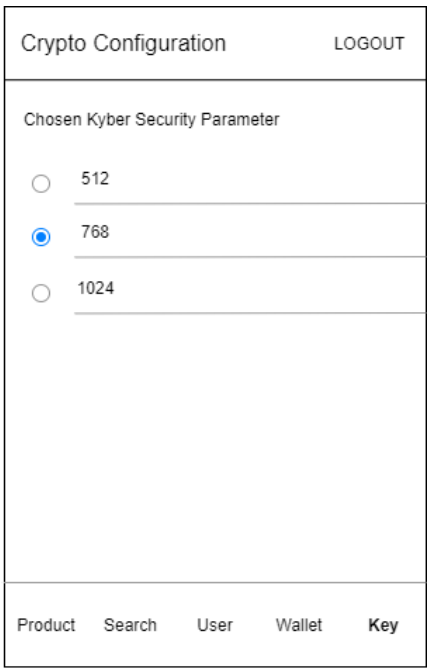

FIGURE 2. Kyber key size configuration

```
7  // Crypto Modules
8  const {Kyber_Encrypt, Kyber_Decrypt} = require('./crystals-kyber/index.js');
9  const Crypto = require('crypto');
10
11 // Kyber Wallet
12 const fire = require('./fire');
13 const db = fire.firestore();
```

FIGURE 3. Hybrid encryption configuration code

CRYSTALS-Kyber library initiation is performed on the Kyber_Encrypt variable and Kyber_Decrypt. Kyber_Encrypt generates a symmetric key randomly and simultaneously performs encapsulation to get the cipher key from the symmetric key. Kyber_Decrypt is used to decapsulate the cipher key into a symmetric key. This symmetric key is used for symmetric encryption using AES-256. AES-256 can be used after initializing the crypto library on the Crypto variable. Kyber_Encrypt and Kyber_Decrypt require input data from a public key and a private key stored in Firestore. Therefore, the Firestore library is initialized on the fire and db variables.

The encryption function is used to perform data encryption on data accepted transactions. The transaction data received consists of two types of data: product data and token data stored in the objectType variable. The symmetric key of the product data can be encapsulated using three different key sizes. In contrast, the symmetric key of the token data can only be encapsulated using a key size of 768. In addition, this function also accepts key size input that is used in the key encapsulation process when calling this function. The encryption process is carried out by calling the Crypto.createCipher library to cipher.final. When calling the Crypto.createCipher library with the AES-256 symmetric encryption algorithm, a symmetric key is required as a buffer. The symmetric key used in symmetric encryption is the output of this.encap function call.

Products that have been encrypted are returned to the function that calls this Encryption function in the form of a JSON Object. The returned JSON object has several key-value pairs, namely key detail containing the ciphertext of the encrypted product, key

cipherKey containing the ciphertext of the symmetric key used in the AES-256 encryption process, key objectType containing the data type of the key detail, and key keySize which contains the key size used in the encapsulation process.

The Decryption function is used to perform data decryption on data transactions. The Decryption function accepts two inputs: transaction data in the form of a Javascript object on the data variable and the key size used to decapsulate the cipher key on the keySize variable. Transaction data in the form of a Javascript object contains detailed key-value pairs, cipherKey, and objectType. The decapsulation function is called to get the symmetric key from the data.cipherKey using the key size that is previously used for encapsulation. The obtained symmetric key is to be used again to decrypt the data.detail to get the plaintext of the transaction data.

The Encapsulation function performs key encapsulation on the symmetric key, later reused in AES-256 symmetric key encryption. The Encapsulation function receives input a variable keySize containing the key size of the public key used in the creation process and the encapsulation process of the symmetric. The Firestore database is called to retrieve the public key with the specified key size. This public key is the input to the Kyber_Encrypt function call along with the keySize variable. The Kyber_Encrypt function is a library from CRYSTALS-Kyber which generates a random symmetric key, and the result of its encapsulation is a cipher key. This symmetric key and cipher key are stored in an array variable cipher_symkey where the cipher key is at index 0, and the symmetric key is at index 1.

The Decapsulation function performs key decapsulation on the cipher key to generate its symmetric key. The decapsulated symmetric key is used in the decryption process using the AES-256 algorithm. The Decapsulation function accepts two inputs, namely the cipher key and key size. The Firestore database is called to retrieve the private key with the key size, which is the input for this decapsulation function. After the private key is obtained, the Kyber_Decrypt function is called a library from CRYSTALS-Kyber. The Kyber_Decrypt function is used to decapsulate the cipher key into a symmetric key. The result of decapsulation in the form of a symmetric key is returned to another function that calls this decapsulation function.

The KeyGeneration function generates public and private keys displayed in Figure 4. The public and private keys are created by calling the Kyber_KeyGen function, which receives an input in the form of the key size to be used. Three key sizes are supported by the CRYSTALS-Kyber library, namely key size 512, key size 768, and key size 1024, which are stored in an array named keySize. Based on Oricon configurations, the KeyGeneration function is called once when the Oricon API server is turned on. When the server starts up, three pairs of public and private keys are generated and stored in the Firestore database in a collection called kyber-keys. The public and private key pairs are pk-K512, sk-K512, pk-K768, sk-K768, pk-K1024, and sk-K1024, where pk represents the public and sk represents the private key.

The two functions of hybrid encryption described previously are implemented in several smart contract functions of the Oricon product authentication application. The two functions are the Encryption function and the Decryption function. In the createProduct function, the transaction data is encrypted before the transaction data is saved to the ledger. In other words, transaction data stored in the ledger is no longer in plaintext but ciphertext. This is so that blockchain service providers cannot see the transaction data. The readProduct function retrieves transaction data in ciphertext form from the ledger to be sent back to the user. Before being returned to the user, the ciphertext is decrypted first by calling the Decryption function to get the plaintext. The implementation of

```
const {Kyber_KeyGen} = require('../../contract/lib/crystals-kyber');
const fire = require('./fire.js');
const db = fire.firestore();

async function main() {
    try {
        const keySize = [512, 768, 1024];
        for(let iterator = 0; iterator < keySize.length; iterator++) {
            let pk_sk = Kyber_KeyGen(keySize[iterator]);
            let pk = pk_sk[0];
            let sk = pk_sk[1];

            await db.collection(`kyber-key`).doc(`pk-K${keySize[iterator]}`).set({
                pk: pk
            });
            console.log('Saved admin K', keySize[iterator],' public key into the firebase!');
            await db.collection(`kyber-key`).doc(`sk-K${keySize[iterator]}`).set({
                sk: sk
            });
            console.log('Saved admin K', keySize[iterator],' public key into the firebase!');
        }
        process.exit(1);
    } catch (error) {
        console.error(`Failed to generate kyber key: ${error}`);
        process.exit(1);
    }
}

main();
```

FIGURE 4. KeyGeneration function code

the Encryption function and the Decryption function is also carried out in several other functions in the smart contract.

3.4. **Testing.** The system is tested using the black-box approach. This test aims to ensure that the created system is running correctly. Blackbox testing is done by calling the API through software called Insomnia. While creating a product, Oricon requires product details such as product code, name, price, origin, release date, description, image, and key size to be included in the JSON body. The key size determines the size of the key used in the encryption data process. The POST method sends "create product" requests from Oricon's frontend application to the smart contract. API endpoint to send "create product" request uses api/create-product. Retrieving a product from the ledger uses the GET method with API endpoint api/product/{product_code}/{key_size}. Product code and key size need to be included in the endpoint to specify which product to retrieve, and the key size used to decrypt the product.

3.5. **Evaluation.** Cryptographic time and smart contract performances are two factors used in order to evaluate the system. Cryptographic time performance measures the time needed for key generation, encryption, and decryption. In contrast, the evaluation of the performance of the smart contract measures the number of successful transactions, failed transactions, latency, and throughput. The cryptographic evaluation uses a library called Performance-Now on Node.Js, and Performance-Now would calculate the time needed in seconds. There are four variations of data size and three key sizes used to evaluate cryptographic time, which can be seen in Table 3. Therefore, there are a total of twelve experiments, with each experiment repeated a hundred times to get the average result of each experiment.

Table 3. Cryptographic time evaluation

No	Key size	Data size (KB)
1	512	25
2	512	50
3	512	75
4	512	100
5	768	25
6	768	50
7	768	75
8	768	100
9	1024	25
10	1024	50
11	1024	75
12	1024	100

Table 4. Smart contract evaluation

No	Key size	MaxMessageCount	BatchTimeout (s)	Number of workers
1	512	20	2	2
2	512	20	2	4
3	768	20	2	2
4	768	20	2	4
5	1024	20	2	2
6	1024	20	2	4

Smart contract performance evaluation uses Hyperledger Caliper with some parameters set as default, such as MaxMessageCount of a maximum of 20 messages and BatchTimeout of a maximum of two seconds. There are two variations in the number of workers being used, that is, two and four. This evaluation uses a smart contract that has implemented the three key sizes of CRYSTALS-Kyber, where the three key sizes follow the sizes set on CRYSTALS-Kyber. In other words, six experiments are carried out, which can be seen in Table 4.

4. **Results and Analysis.** Based on the system testing results using blackbox testing, it appears that the system is running as expected. Blackbox testing results can be seen in Table 5.

4.1. **Evaluation of key generation time comparison for all key sizes.** Based on Figure 5, it can be seen that the time required for key size 512 to perform key generation is shorter than the other key sizes, which is 0.0036 seconds, while key size 1024 requires the longest key generation time, which is 0.0110 seconds. The difference in time required to perform key generation between key size 768 and key size 1024 experienced a more significant difference, which is 0.0054 seconds, compared to the difference between key size 512 and key size 768, which is 0.0020 seconds.

4.2. **Evaluation of encryption and decryption time comparison for all key sizes.** Figure 6 and Figure 7 show respectively a comparison graph of encryption time and decryption time with different data sizes for all key sizes.

TABLE 5. Test results

No	Test scenario	Expected results	Test results	Conclusion
1	The user encrypts data using key size 512	The response from the API states that the data is successfully encrypted with a key size of 512, returning encrypted data	Status "Successfully encrypt a product with key size 512", returns encrypted data	Valid
2	The user encrypts data using key size 768	The response from the API states that the data is successfully encrypted with a key size of 768, returning encrypted data	Status "Successfully encrypt a product with key size 768", returns encrypted data	Valid
3	The user encrypts data using key size 1024	The response from the API states that the data is successfully encrypted with a key size of 1024, returning encrypted data	Status "Successfully encrypt a product with key size 1024", returns encrypted data	Valid
4	The user encrypts data without providing information about the key size to be used	The response from the API states that the data is successfully encrypted with the default key size of 768, returning encrypted data	Status "Successfully encrypt a product with key size 768", returns encrypted data	Valid
5	The user encrypts data using a key size other than that supported by the CRYSTALS-Kyber algorithm (512, 768, and 1024)	The response from the API states that the data failed to be encrypted	Status "There is a problem encrypting the product"	Valid
6	The user decrypts the ciphertext using the same key size as the key size used in the data encryption process	The response from the API states that the data has been successfully decrypted with the key size x, where x is the key size given by the user	Status "Successfully decrypt a product with key size x", returns plaintext of the data	Valid
7	The user decrypts the ciphertext without providing information about the key size to be used	The response from the API states that the data is successfully decrypted with the default key size of 768	Status "Successfully decrypt a product with key size 768", returns plaintext of the data	Valid
8	The user decrypts the ciphertext using a key size that is different from the key size used in the data encryption process	The response from the API states that the data failed to be decrypted	Status "There is a problem decrypting the product"	Valid
9	The user calls the readProduct function using the ciphertext, cipherkey, and key size generated from the createProduct function	The result of the decrypted ciphertext has the same value as the plaintext before it is encrypted	The result of the decrypted ciphertext is the same as the plaintext before it is decrypted	Valid

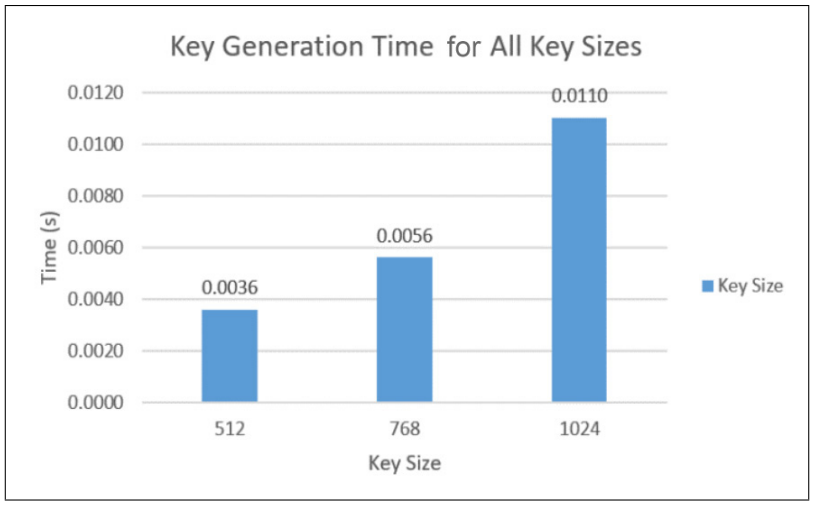

FIGURE 5. Comparison graph of key generation time for all key sizes

Figure 6 and Figure 7 show that the key size 1024 proved to take the longest time to perform the encryption and decryption process compared to the other two key sizes, namely around 0.0698 to 0.0721 seconds to encrypt data and around 0.0783 to 0.0813 seconds to decrypt data. Key size 512 has the fastest encryption and decryption time, but increasing time tends to be unstable. It differs from the key size 768, which tends to experience a steady increase in time, where the time required is longer as the size of the data used increases. For better understanding, some terms being used are explained as the followings.

1) Smart contract plain, used in this research to refer to smart contracts that do not implement any key size used in previous work.
2) Smart contract 512, used in this research to refer to smart contracts implemented with key size 512.
3) Smart contract 768, used in this research to refer to smart contracts implemented with key size 768.

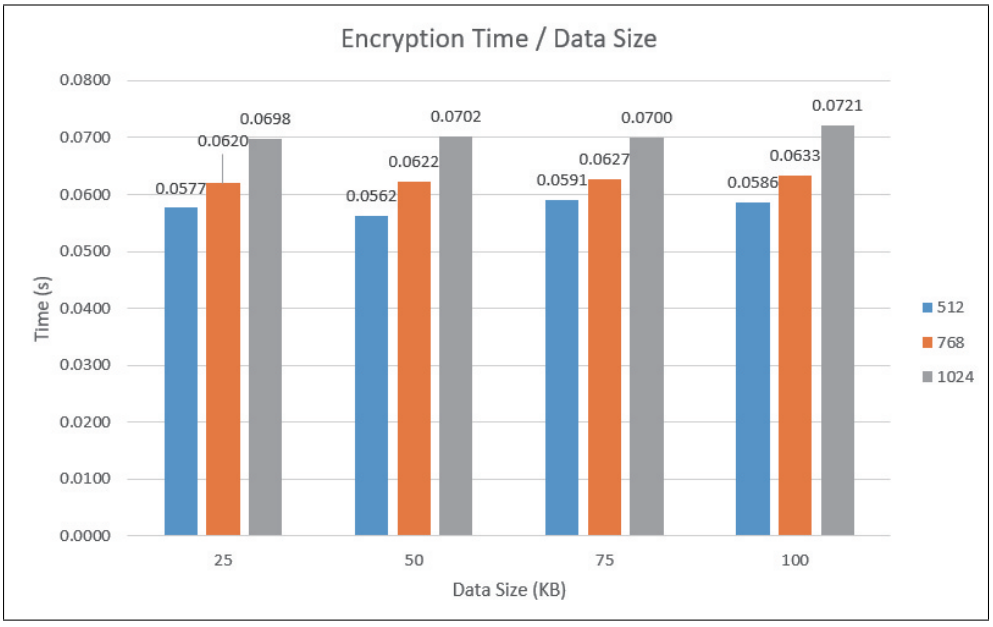

FIGURE 6. Comparison graph of encryption time with data size for all key sizes

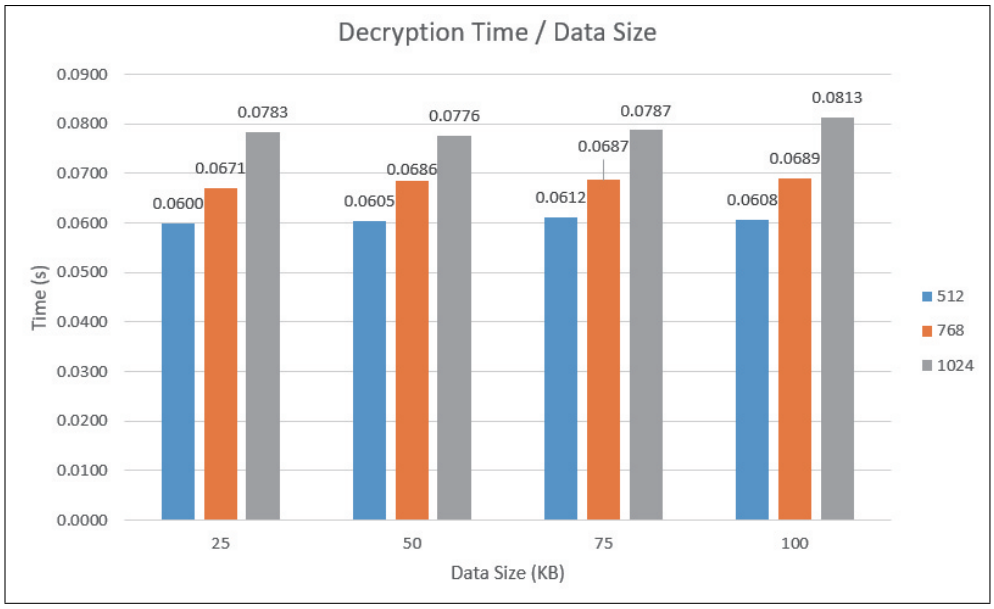

FIGURE 7. Comparison graph of decryption time with data size for all key sizes

4) Smart contract 1024, used in this research to refer to smart contracts implemented with key size 1024.

4.3. **Evaluation of successful transaction.** Based on the comparison graph of successful transactions given in Figure 8, it can be seen that smart contracts plain has the highest number of successes in transactions sent to both variations of the number of workers. Smart contract 768 ranks second in terms of the highest number of transactions successfully sent to the ledger, followed by smart contract 1024 and smart contract 512. It can

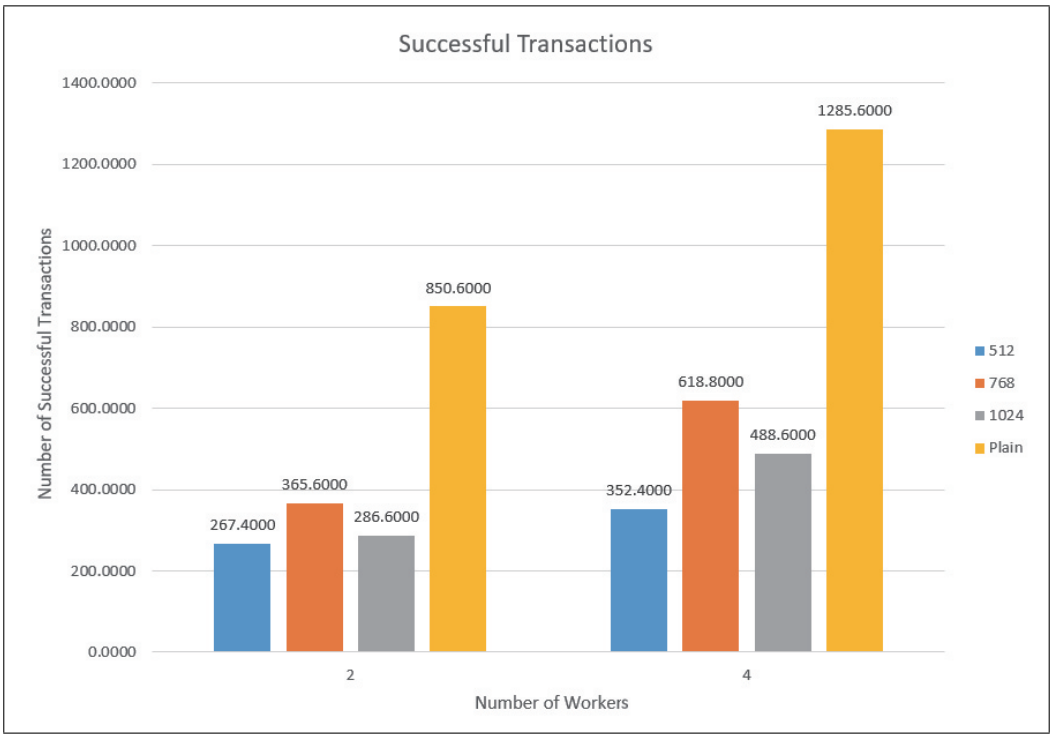

FIGURE 8. Comparison graph of successful transactions on smart contracts

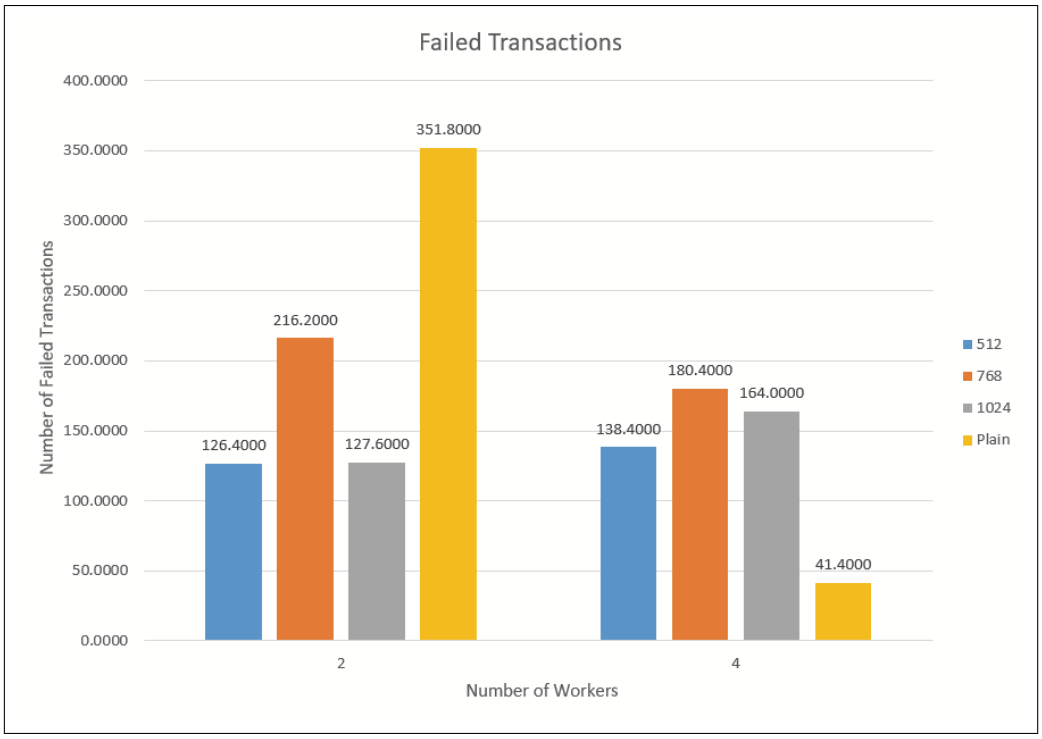

FIGURE 9. Comparison graph of failed transactions on smart contracts

also be seen from the graph that the number of workers does not affect the performance of a key size implemented in smart contracts, where smart contracts 768 is a smart contract with a key size that has the highest number of successes compared to smart contracts 1024 and smart contracts 512 at any variation in the number of workers. The same can be seen in the smart contract 1024, which is better than smart contract 512 in any variation of the number of workers.

4.4. **Evaluation of failed transaction.** Figure 9 shows that the number of failed transactions using two workers on a smart contract plain is inversely proportional to a smart contract plain that uses four workers. The smart contract plain has the highest number of failed transactions when using two workers simultaneously, and it is also the smart contract with the lowest failed transaction when using four workers. The four workers test shows that the smart contract 768 has the highest number of failed transactions compared to other smart contracts. However, there is no significant difference between several failed transactions between smart contracts 512, smart contracts 768, and smart contracts 1024.

4.5. **Evaluation of latency.** A significant latency indicates that the time it takes to make one transaction on the smart contract is getting longer, thus showing that the performance of the smart contract is getting worse, as displayed in Figure 10. Smart contract 512 using four workers has the most significant latency, followed by smart contract 1024. For two workers, smart contract 768 has the most significant latency. Even so, the difference in latency of those smart contracts that apply key size 512, key size 768, and key size 1024 is not too significant, which is only a difference of 0.474 seconds to 0.576 seconds. The smart contract 768 performance proved better than the other three smart contracts with four workers.

4.6. **Evaluation of throughput.** Smart contracts plain performs the best when using either two or four workers. This can be seen in Figure 11 from the smart contract plain,

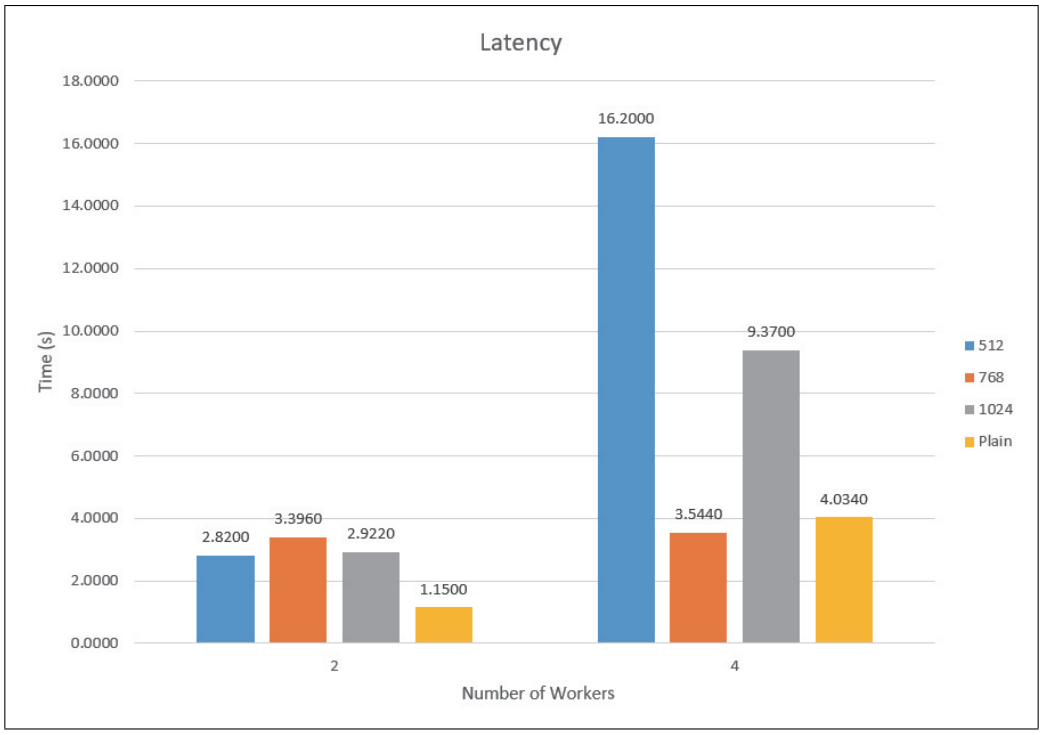

FIGURE 10. Comparison graph of latency on smart contracts

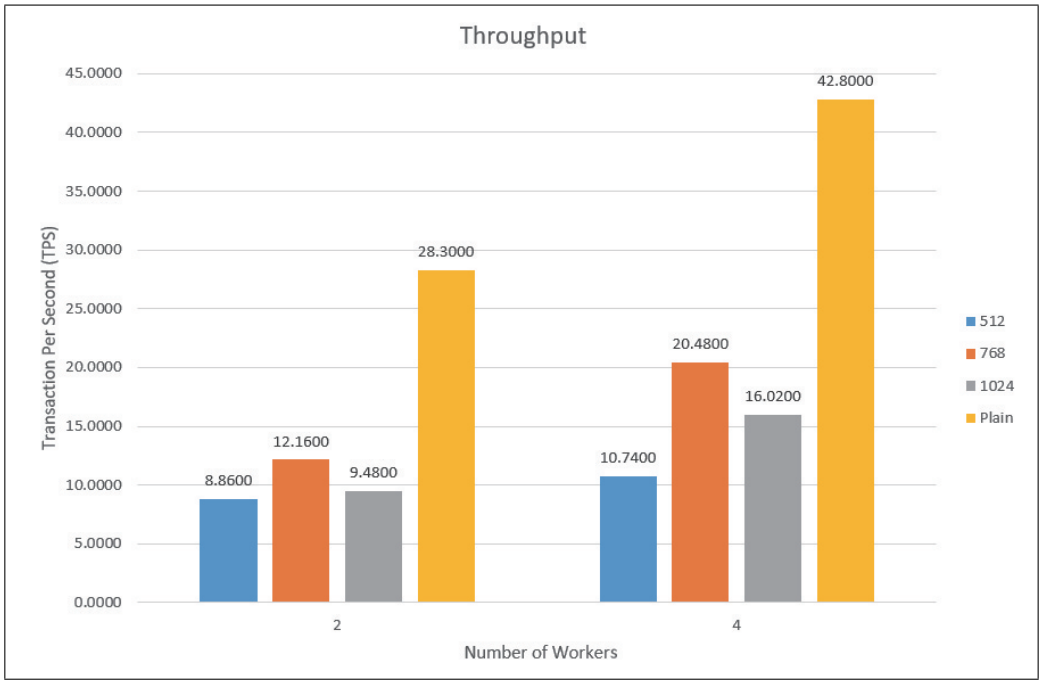

FIGURE 11. Comparison graph of throughput on smart contracts

which has the largest amount of throughput, which is 28.3 TPS when using two workers and 42.8 TPS when using four workers. Meanwhile, for smart contracts implemented with key sizes, it can be seen that smart contracts 768 have a higher amount of throughput than the other two key sizes. While using four workers, the throughput of smart contracts 768 is almost twice the amount of throughput of smart contracts 512. The use of key size in smart contracts has an impact on reducing the amount of throughput compared

to plain smart contracts. Decrease in the amount of throughput on smart contracts 512, smart contracts 768, and smart contracts 1024, is presented as the following.

1) Smart contracts 512's performance is decreased by −68.69% when using two workers and −74.91% when using four workers.
2) Smart contracts 768's performance is decreased by −57.03% when using two workers and −52.15% when using four workers.
3) Smart contracts 1024's performance is decreased by −66.50% when using two workers and −62.57% when using four workers.

5. **Conclusions.** This study finds that the hybrid encryption scheme of CRYSTALS-Kyber and AES-256 is a feasible and working solution for blockchain applications to withstand quantum attacks. The scheme is implemented and tested on the Oricon blockchain application as proof of concept and demonstration. The aim is to show that this proposed approach requires minimum to no modifications to the blockchain applications. This study has proven this and benefits other existing blockchain applications to adopt this hybrid encryption scheme for the post-quantum era.

The addition of the hybrid encryption scheme on Oricon has been tested using the black-box methodology to prove and validate the functionalities. Performance evaluation of the work is another primary concern and contribution of this study, and the addition of security measures is costly to the system resources. Thus, observation of the performance changes of the application system is essential to find the best configuration of the hybrid encryption scheme and to measure the changes in the application performance.

The CRYSTALS-Kyber 768 is the most optimal choice considering the application performance compared to the other two key sizes. This conclusion is drawn from the evaluation that shows that the increase in encryption and decryption time tends to be stable compared to the other two key sizes when the data is getting larger. In addition, it delivers a peak number of successful transactions among other key size choices in two and four workers configurations. However, due to its peak throughput, its failed transactions are also the highest compared to the other key size choices. The throughput is measured at 12.16 TPS for two workers and 20.48 TPS for four workers. In addition, the latency is considered low compared to the two other key sizes, with 3.396 seconds for two workers and 3.544 seconds for four workers.

Although the most optimal key size, implementing key size 768 decreases the amount of throughput generated compared to the original application throughput. Adding a post-quantum security module is crucial for the blockchain application to withstand quantum attacks. This study measures the performance changes of the application, which is a drop of 57.03% when using two workers and 52.15% when using four workers. Again the best results obtained from this study compared to the other two key sizes, which are for 512 and 1024 respectively, are −68.69% for two workers, −74.91% for four workers, and −66.50% for two workers and −62.57% for four workers.

Future work based on this study is to measure the complexity of the added hybrid encryption system to the Oricon blockchain application and to measure the performance drops in more workers' configurations. It is also recommended to apply the hybrid encryption scheme in this study to other blockchain applications to observe and compare the results. The use of CRYSTALS-Kyber in this study could be evaluated and changed to more versatile and robust post-quantum cryptography published in the future.

Acknowledgment. The authors would like to thank Universitas Multimedia Nusantara for the support of this research work.

REFERENCES

[1] Simplilearn, *What Is Data Encryption: Types, Algorithms, Techniques and Methods*, 2022, https://www.simplilearn.com/data-encryption-methods-article, Accessed on Jul. 07, 2022.

[2] J. Thakkar, *Types of Encryption: 5 Encryption Algorithms & How to Choose the Right One*, 2020, https://securityboulevard.com/2020/05/types-of-encryption-5-encryption-algorithms-how-to-choose-the-right-one/, Accessed on Jul. 07, 2022.

[3] H. Saini, *8 Strongest Data Encryption Algorithms in Cryptography*, 2022, https://www.analyticssteps.com/blogs/8-strongest-data-encryption-algorithms-cryptography, Accessed on Jul. 07, 2022.

[4] P. Thorsteinson and G. G. A. Ganesh, RSA: The most used asymmetric algorithm, *NET Security and Cryptography*, 2003.

[5] Arcserve, *5 Common Encryption Algorithms and the Unbreakables of the Future*, 2022, https://www.arcserve.com/blog/5-common-encryption-algorithms-and-unbreakables-future, Accessed on Jul. 07, 2022.

[6] E. Gerjuoy, Shor's factoring algorithm and modern cryptography. An illustration of the capabilities inherent in quantum computers, *Am. J. Phys.*, DOI: 10.1119/1.1891170, 2005.

[7] A. W. Wicaksono and A. Wicaksana, Implementation of Shor's quantum factoring algorithm using projectQ framework, *Int. J. Eng. Adv. Technol.*, vol.8, no.6 Special Issue 3, DOI: 10.35940/ijeat.F1009.0986S319, 2019.

[8] A. Wicaksana, Anthony and A. W. Wicaksono, Web-app realization of Shor's quantum factoring algorithm and Grover's quantum search algorithm, *TELKOMNIKA (Telecommunication Comput. Electron. Control)*, vol.18, no.3, 1319, DOI: 10.12928/telkomnika.v18i3.14755, 2020.

[9] A. Baumhof, *Breaking RSA Encryption – An Update on the State-of-the-Art*, 2019, https://www.quintessencelabs.com/blog/breaking-rsa-encryption-update-state-art/, Accessed on Apr. 22, 2022.

[10] C. Gidney and M. Ekerå, How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits, *Quantum*, vol.5, p.433, 2021.

[11] M. M. El Khatib, F. Beshwari, M. Beshwari and A. Beshwari, The impact of blockchain on project management, *ICIC Express Letters*, vol.15, no.5, pp.467-474, 2021.

[12] L. Mark, V. Ponnusamy, A. Wicaksana, B. B. Christyono and M. Widjaja, A secured online voting system by using blockchain as the medium, in *The Smart Cyber Ecosystem for Sustainable Development*, P. Kumar, V. Jain and V. Ponnusamy (eds.), Wiley, 2021.

[13] B. B. A. Christyono, M. Widjaja and A. Wicaksana, Go-Ethereum for electronic voting system using clique as proof-of-authority, *TELKOMNIKA (Telecommunication Comput. Electron. Control)*, vol.19, no.5, 1565, DOI: 10.12928/telkomnika.v19i5.20415, 2021.

[14] M. Allende et al., *Quantum-Resistance in Blockchain Networks*, DOI: http://dx.doi.org/10.18235/0003313, 2021.

[15] W. Philips and A. Wicaksana, Hybrid approach of quick response code and non-fungible token in private permissioned blockchain for anti-counterfeiting, *International Journal of Innovative Computing, Information and Control*, vol.18, no.5, pp.1617-1632, DOI: 10.24507/ijicic.18.05.1617, 2022.

[16] Hyperledger Caliper, *Getting Started*, https://hyperledger.github.io/caliper/v0.3.2/getting-started/, Accessed on Apr. 22, 2022.

[17] NIST, *Post-Quantum Cryptography PQC*, 2017, https://csrc.nist.gov/projects/post-quantum-cryptography, Accessed on Apr. 22, 2022.

[18] NIST, *PQC Standardization Process: Third Round Candidate Announcement*, 2020, https://csrc.nist.gov/News/2020/pqc-third-round-candidate-announcement, Accessed on Apr. 22, 2022.

[19] R. Avanzi et al., *CRYSTALS-Kyber Algorithm Specifications and Supporting Documentation (Version 3.01)*, https://pq-crystals.org/kyber/data/kyber-specification-round3-20210131.pdf, 2021.

[20] P. Patil and R. Bansode, Performance evaluation of hybrid cryptography algorithm for secure sharing of text & images, *Int. Res. J. Eng. Technol.*, vol.7, no.9, pp.3774-3778, 2020.

[21] Quantum Resistant Ledger, *The Future of Post-Quantum Resistant Blockchains*, 2022, https://www.theqrl.org/the-future-of-post-quantum-resistant-blockchains/, Accessed on Jul. 09, 2022.

[22] T. M. Fernández-Caramés and P. Fraga-Lamas, Towards post-quantum blockchain: A review on blockchain cryptography resistant to quantum computing attacks, *IEEE Access*, vol.8, pp.21091-21116, DOI: 10.1109/ACCESS.2020.2968985, 2020.

[23] P. Zeng, S. Chen and K.-K. R. Choo, An IND-CCA2 secure post-quantum encryption scheme and a secure cloud storage use case, *Human-Centric Comput. Inf. Sci.*, vol.9, no.32, 2019.

[24] M. R. Albrecht, C. Cid, K. G. Paterson, C. Tjhai and M. Tomlinson, *NTS-KEM – Round 2 Submission*, https://csrc.nist.gov/CSRC/media/Presentations/nts-kem-round-2-presentation/images-media/nts-kem.pdf, 2019.
[25] Hyperledger Foundation, *Hyperledger Caliper*, https://www.hyperledger.org/use/caliper, Accessed on Apr. 22, 2022.

Author Biography

Kevin Hendy received the S.Kom., CEH degree in informatics engineering from Universitas Multimedia Nusantara in 2021. He is currently working as a Software Engineer for a company focusing on digital payment. His current research interests include quantum cryptography, cryptocurrency, and non-fungible token.

Arya Wicaksana is a lecturer at the Department of Informatics at UMN. He received Master Degree in VLSI Engineering from Universiti Tunku Abdul Rahman. He successfully demonstrated the UTAR first-time success ASIC design methodology on a multi-processor system-on-chip project using 0.18 μm processing technology in 2015. His main research interests are blockchain applications and computational intelligence. He recently worked on a decentralized autonomous social media. He is affiliated with ACM and IEEE as a professional member. He has served as an invited reviewer in IEEE ACCESS, IJNMT, and IFERP and an invited author in IntechOpen and other scientific publications.