applied sciences

Article

Block Data Record-Based Dynamic Encryption Key Generation Method for Security between Devices in Low Power Wireless Communication Environment of IoT

Soohwan Cho [1,2], Deokyoon Ko [3] and Sooyoung Park [1,2,*]

1 Department of Computer Science & Engineering, Sogang University, Seoul 04107, Korea; soohwan@emblock.co.kr
2 Department of R&D, EMBLOCK Inc., Seoul 04074, Korea
3 NonceLab. Inc., 802 Seoul Blockchain Center, 78, Mapo-daero, Mapo-gu, Seoul 04168, Korea; dykoh@noncelab.com
* Correspondence: sypark@sogang.ac.kr

Received: 15 September 2020; Accepted: 31 October 2020; Published: 9 November 2020

Featured Application: Sensor data record-based dynamic encryption key technology which applies the mechanism of blockchain for security between devices in the low-power wireless communication environment of Internet of Things (IoT).

Abstract: The Internet of Things uses low-power wireless communication for wireless connectivity and efficient energy. Low-power wireless communication is applied to IoT for wireless connection and efficient energy consumption in various areas such as wearable devices, smart homes, and power plants in order to send and receive data and control the environment. Security is becoming more important because the Internet of Things controls real physical systems. For the security of the Internet of Things, the encryption key is important to identify and authenticate devices that are trusted. The static encryption key method used for devices is likely to be calculated in reverse through the value of the key and is vulnerable to exploitation attacks. This requires the application of dynamic encryption keys that generate keys periodically. However, in the case of low-power wireless communication, the asynchronous communication method and the packet loss make it difficult to apply existing dynamic encryption key technologies. In this paper, we proposed dynamic encryption key method that applies the mechanism of the block chain to solve these problems. Based on the history of sensor data between devices, encryption keys are dynamically generated. The proposed method is to generate the same encryption key between devices with only one step of asynchronous communication considering packet loss. The proposed method is also validated in terms of availability and security in the Internet of Things low-power wireless communication.

Keywords: blockchain; Internet of Things; dynamic encryption key; IoT device security; low power wireless communication; Bluetooth low energy

1. Introduction

1.1. Study Background and Motivation

Internet of Things (IoT) refers to intelligent technology and service whereby various sensors and communication technologies such as Bluetooth, Wi-Fi, LTE, Zigbee, and NFC are built-in in a system that can identify physical objects (things) to exchange information between thing and thing, and human and thing by linking data in real time based on the Internet [1]. Low-power wireless communication is applied to IoT for wireless connection and efficient energy consumption in various areas such as wearable devices, smart homes, and power plants in order to send and receive data and control the environment [2,3].

However, as the IoT technology has advanced and the services have expanded, the security issue has become more important. Unlike computer online systems, IoT has the security problem of cyber-physical system (CPS) [4]. CPS security is important because security damages in the virtual space such as computer online system hacking damages can extend to the real physical system. In the worst case, security problems of IoT can be a threat to human lives based on incorrect sensor data values and device controls [5,6].

Identification and authentication of devices are important for the security of IoT. Device identification refers to reliable authentication that can identify corresponding devices and determine their authenticity for safe operation of various devices participating in a network [7,8]. Threat of device identification/authentication can cause problems such as data tampering and malicious control based on unauthorized access to the devices. When identifying and authenticating devices in IoT environment, the most important factor is encryption key technology. Application of a vulnerable encryption key technology leads to a threat to device identification/authentication, causing vulnerability in the whole system security [9,10] the same technology used in the existing computer system is used for the encryption key technology between devices in the IoT environment [7]. Encryption keys are classified into symmetric keys and asymmetric keys depending on whether the encryption and the decryption key are identical or different. In general, when an identical level of security is provided, a symmetric key is used for encryption and decryption because the computing speed using symmetric key is fast [11]. In the IoT environment that requires real-time operation, the symmetric key method is used because of computing performance and computational speed. Furthermore, static symmetric encryption key method is often used whereby a key, once loaded on the device, does not change [11]. However, because such a static key method is used without changing the key, the encryption pattern is continuously exposed and consequently, a possibility exists that the key values can be computed through reverse engineering using the encrypted data. Moreover, if the encryption key is stolen or exposed through the agent that manages the encryption key, the security between the devices becomes futile [12].

Dynamic encryption key technology is required to prevent this. Dynamic encryption key is a method of periodically generating and using new encryption keys by both ends of communication [12]. When the dynamic encryption key method is used, backtracking of the encryption key values is difficult because the key is generated periodically; furthermore, even if the current key is stolen, the key generated next cannot be predicted. If such a dynamic encryption key technology is applied between IoT devices, the security improves compared to the static key method [13–15].

1.2. Definition of Problems

For IoT, low-power wireless communication environment is often used, which can operate the devices long time, considering the energy efficiency [3]. Bluetooth low energy (BLE) is a typical example [16]. Low-power Bluetooth communication is used between devices in various areas in order to send, receive, and control sensor data. However, the following problems exist when the dynamic encryption key technology providing excellent security is applied to low-power wireless communication environment.

1.2.1. Availability

Encryption Key Generation Problem

In some cases, dynamic encryption key technology uses a synchronous communication method to regenerate the key [17,18]. The newly generated encryption key's generation information or generation request is generated through the request and response processes between both ends of communication. However, asynchronous communication (For example, ACL Asynchronous Connection-Less) between devices is used in the low-power wireless communication environment of IoT [19]. A sensor device sends or propagate the environment sensor data periodically to other sensor devices or gateway device; the devices that receive them do not send the response. Thus, the conventional dynamic encryption key technology that uses synchronous communication is difficult to apply to low-power wireless communication of IoT.

Encryption Key Synchronization Problem

Encryption key synchronization problem refers to a case in which synchronization to a same encryption key fails when the encryption key is changed to the new subsequent key at both ends of communication because the synchronization of same key is not performed properly [17]. In the case of low-power Bluetooth communication, about 2% packet loss occurs depending on the transmission interference between devices or the network structure [20,21]. Encryption key synchronization problem occurs if packet loss occurs when both ends request the key generation information or notification to generate a dynamic encryption key. If the encryption key synchronization problem occurs due to packet loss, the request and response information should be sent and received based on the synchronization method, but this is not suitable for asynchronous communication method.

Performance Problem

In the case of IoT devices that perform low-power wireless communication, data are often transmitted in real-time from a low-performance device. Therefore, the performance of dynamic encryption key technology is another factor that should be taken into consideration. It is not suitable to have the encryption key generation require a high computational complexity. Furthermore, it is difficult to apply if too much time is consumed to generate encryption keys and encrypt and decrypt data in the IoT environment that requires real-time operation.

1.2.2. Security Aspect

The dynamic encryption key technology that has resolved the availability problem in the low-power wireless communication IoT environment mentioned in Section 1.2.1 should also satisfy the security requirement. Users with malicious intent should not be able to guess the subsequent key (randomness) and predict the subsequent key (unpredictability) even if the current key is stolen. The key generation information delivery problem [22] should be prevented when both ends of communication key generate a key.

1.3. Purpose of Study

This paper shares the following purposes to solve the dynamic encryption key application problem in the low-power wireless communication environment mentioned in Section 1.2.2. This paper proposes a method to use communication at least once based on the asynchronous communication method when generating a dynamic encryption key between devices, generating the same encryption key based on the asynchronous method even when packet loss occurs. The dynamic encryption key generated using the proposed method is unpredictable, and even if the current key is stolen, the new subsequent key cannot be guessed, thereby satisfying the security requirement. This paper proposes a dynamic encryption key method that can be used on low-performance devices.

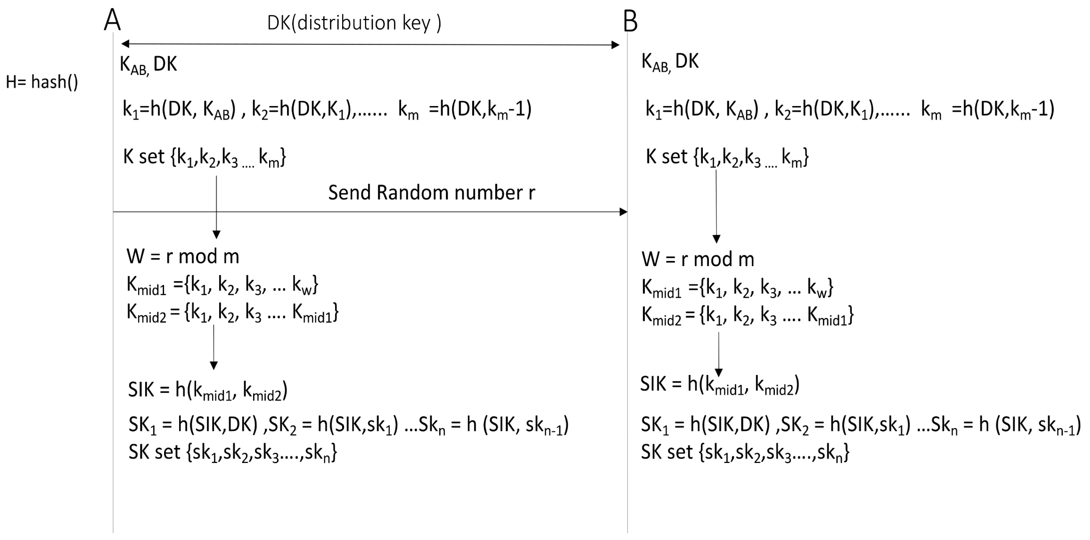
The approach of this paper applies the mechanism of blockchain to the dynamic encryption key generation. The study by Soohwan et al. [23] generated unpredictable hash values dynamically based on the IoT sensor data; using this, proposed a consensus algorithm that selects block generating nodes. The study by WooSeung [24] generated unpredictable hash values dynamically based on the transaction history generated by microcontroller units (MCUs) in controller area network (CAN) protocol environment for security; using this, proposed a method of generating a new dynamic encryption key. The mechanism of blockchain applied in the related studies [23,24] employs a method of dynamically generating unpredictable hash by transaction history. The newly generated hash value can be known if and only if all the details are known from the not tampered first transaction to the current one. In this paper, we also employ such mechanism of blockchain [25–27] to generate block data records based on the sensor data between devices. The difference from the related work [23,24] is that a new dynamic encryption key is generated through a dynamically generated hash value based on the sensor history between IoT devices. Furthermore, we propose a method to generate a same encryption key by asynchronous communication method using the TargetValue and the FrequencyTable based on the sensor data history.

2. Related Work

In this chapter, we investigate the existing studies on dynamic encryption key, and find out the applicability in low-power wireless communication environment through a comparison table.

2.1. Dynamic Encryption Key of OTP (S/KEY) Method

The OTP (SKEY) method [28] is an authentication key generation method developed by Bell Communications Research and is used for authentication in UNIX based operating systems. In this method, the generation algorithm sends a random secret key determined by the client to the server. The secret key received from the client is used as the first value. Then, the task of obtaining the hash value for the previous result value is repeated N times based on the hash chain method. The generated N OTPs are stored in the server. The hash function determined by the client is applied n-i times and sent to the server. The server applies the hash function to the value received from the client once and checks whether the result matches the n-I + stOTP stored in the server. In the study of Limited-Used Key Generation Scheme [29], a method is proposed for generating a one-time password key by applying the OTP (S/KEY) method. This scheme generates an encryption key sequence in advance to generate a dynamic key. Because each encryption key is used to encrypt only one message, every message uses a different encryption key. This scheme uses the predefined hash function and the shared K_{AB} master key. In this scheme, a dynamic encryption key is generated as shown in Figure 1.



A
DK(distribution key)
B
H= hash()
K_{AB}, DK
k_1=h(DK, K_{AB}) , k_2=h(DK,K_1),...... k_m =h(DK,k_m-1)
K set {k_1,k_2,k_3 .... k_m}
Send Random number r
W = r mod m
K_{mid1} ={k_1, k_2, k_3, ... k_w}
K_{mid2} = {k_1, k_2, k_3 .... K_{mid1}}
SIK = h(k_{mid1}, k_{mid2})
SK_1 = h(SIK,DK) ,SK_2 = h(SIK,sk_1) ...Sk_n = h (SIK, sk_{n-1})
SK set {sk_1,sk_2,sk_3....,sk_n}

Figure 1. Limited-Used Key Generation Scheme.

The authentication server generates distribution key (DK) and sends it using the authenticated key exchange protocol. The client and the authentication server generate the basic setting keys, K set ({K_1 = hash(DK, K_{AB}), K_2 = h(DK, K_1)... K_m = h(DK, K_{m-1})}) through iterative hash of DK and K_{AB}. Subsequently, the authentication server generates random number r and sends it to the client, then w = r mod n is derived from the random number r. The authentication server and the client select one K_{mid1} among $\{K_1, K_2, K_3....K_w\}$ from w, and select K_{mid2} among $\{K_1, K_2, K_3....K_{mid1}\}$. The authentication server and the client obtain SIK = $h(k_{mid1}, K_{mid2})$ and generate the session key (SK) sets as follows through iterative hash of SIK and DK: SK_1 = h(SIK,DK), SK_2 = h(SIK,SK_2).... SK_n = h(SIK,SK_{n-1}).

In the OTP (S/KEY) method, there is no recovery plan protocol for solving the encryption key synchronization problem. In the current method, two or more communication steps of synchronization method is required if the key synchronization fails due to packet loss. A and B should send the request/reply messages to inform the encryption key mismatch and attempt to match the current counter. In the case of OTP method, because the initial seed value of encryption key is stored in the OTP server, it is vulnerable to hijacking attacks [24]. If a malicious attacker steals the seed value of OTP server and reproduces the mechanism of encryption key generation, the key can be predicted. Furthermore, the OTP method employs a method of generating the encryption keys that will be used in advance. Therefore, encryption keys have to be regenerated when the encryption keys are all exhausted.

2.2. Session Key of Diffie–Hellman Method

Diffie–Hellman method's session key [30,31] solves the key delivery problem and is a highly stable. Thus, this dynamic encryption key method is frequently used. This method solves the key delivery problem by using discrete logarithm to synchronize the same dynamic encryption key at both ends. The method of Diffie–Hellman is carried out in the manner shown in Figure 2 below. A uses the private key a to generate g^a mod P (asymmetric key of a). B also uses the private key b to generate g^b mod p (asymmetric key of b). A sends g^a mod P to B, and B sends g^b mod p to A. Finally, A uses its own private key a to generate g^{ab} mod p, and B uses its own private key to generate gab mod p. A and B use the newly generated key as the same symmetric key (secret key). The attacker cannot generate the same symmetric key even if the attacker were to obtain the key generation information.

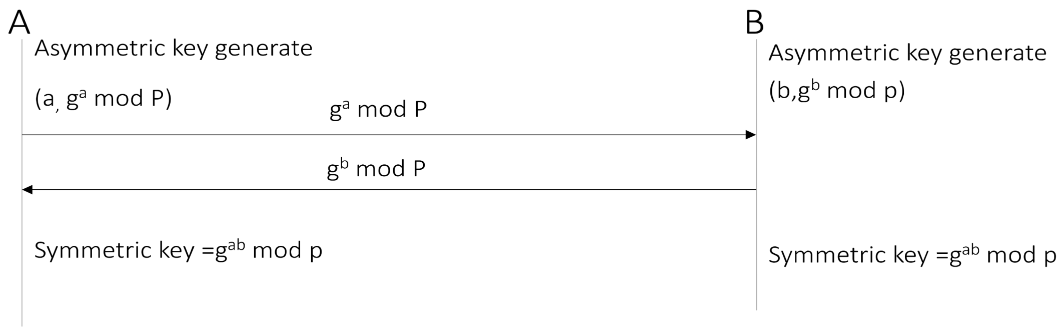


Figure 2. Diffie–Hellman's key generation method.

Diffie–Hellman method uses a same key in a session and generates a new key through safe random number algorithm for the next session. However, Diffie–Hellman method requires at least two steps of synchronous communication in order to generate a new key. Moreover, it uses a discrete logarithm-based asymmetric key method. therefore, the availability decreases due to large computational cost and time consumption. When the encryption key synchronization fails due to packet loss, two or more steps of synchronous communication are required. For generation of a new session key, a safe random number generation algorithm is required to generate a seed value of the key. However, if the random number generation algorithm used is exposed or identified, the encryption key can be predicted.

2.3. Key Generation Algorithm Based on Shared Message History for In-Vehicle Security Network (Blockey)

This key generation algorithm method (blockey) employs a dynamic encryption key generation method applying the mechanism of blockchain for in-vehicle security [24]. Inside the vehicle, electric control units (ECUs) communicate in the form of bus using the CAN protocol. Figure 3 shows the method of blockey. In CAN communication, ECUs send and receive the generated transactions and accumulate the message history up to a certain baseline value; afterwards, the previous block's hash value and the message history's hash value are used to generate a new block. In this encryption key method, the transaction histories occurring at ECU are used to generate a new hash value, which is then used as the seed value of key to generate the key dynamically. The history-based block key algorithm generates a key without requiring a random number generation algorithm in generating the key. Furthermore, it does not require management because the data value by external environment are continuously used as the seed values of key. All transaction history must be known to be able to guess the subsequent encryption key. Even if the current key is stolen, all previous transaction details must be acquired in order to be able to guess the key generated next.

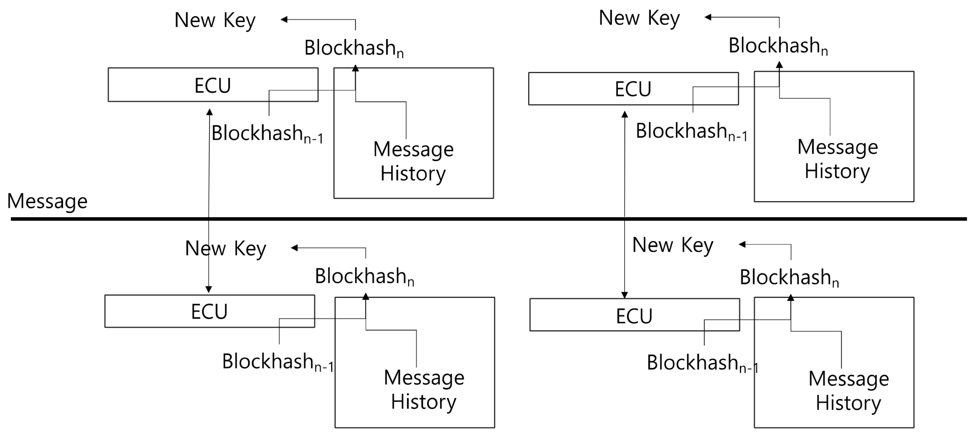


Figure 3. History-based encryption key algorithm.

However, all messages have to be maintained the same to synchronize the same encryption key. Compared to CAN protocol, the low-power wireless communication environment is prone to have loss of message more often. Therefore, mismatch can occur often in the encryption key synchronization. In order to recover the encryption key, all messages should be made identical based on the synchronous communication method. Since this is performed through two or more steps by the synchronous communication method, it is not suitable.

2.4. Comparison of Related Work

The following comparison Table 1 is used to examine the applicability of the related studies discussed earlier in the low-power wireless communication environment of IoT. The comparison is made in the availability aspect and the security aspect.

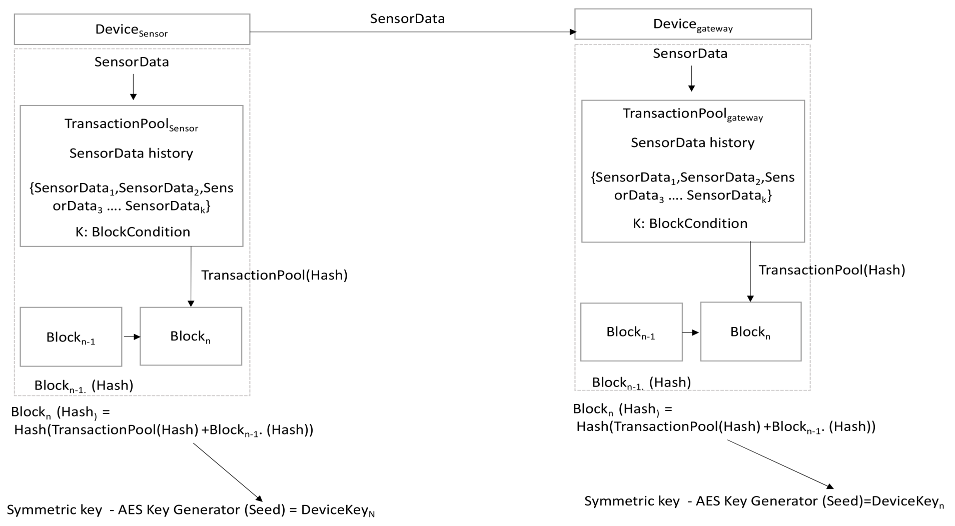
The OTP (S/Key) method satisfies the criteria for the encryption key generation communication method and the required computational cost/time for key generation because the dynamic encryption keys used are generated in advance. However, considering the encryption key recovery communication method used in the case of packet loss, it is not appropriate because it requires performing two or more steps of synchronous communication. If a malicious user steals the initial seed value of OTP server, the user can generate all encryption keys; thus, the security is vulnerable to hijacking attacks. Exposure of key generation information can be prevented because only the counter values are sent when generating an encryption key. As long as the seed value is not stolen, the randomness and unpredictability of encryption key are satisfied. Diffie–Hellman method is not appropriate because two or more steps of communication based on the synchronous method are required to generate and recover an encryption key. Because asymmetric key is used, the required computational cost and time for key generation are high compared to those of other related studies. Because the key generation information is exchanged using the discrete logarithm problem, the exposure of key generation information is prevented. The randomness and unpredictability of encryption key are satisfied on the premise that a safe random number generation algorithm is used. However, if the random number generation algorithm is exposed, the encryption keys can be generated. In the case of block key, the criteria for the encryption key generation communication method and the key generation information exposure prevention are satisfied because the encryption key is generated independently at both ends of communication. However, considering the encryption key recovery communication method, it is not appropriate because two-step communication of synchronous method is required. In the case of hijacking attack, there is no exposure problem of random number generation algorithm and all the history of seed has to be known; thus, the security of block key method is deemed high compared to that of methods of other related studies.

Table 1. Comparison table of related work.

	OTP (S/Key) Kungpisdan et al. [29]	**Diffie–Hellman Diffie and Hellman [30]**	**Blockey WooSeung [24]**
Availability	Encryption Key Generation Communication Method/Steps		
	Asynchronous/1 step	Synchronous/2 steps	Asynchronous/0 step
	Encryption Key Recovery Communication Method/Steps		
	Synchronous/2 steps	Synchronous/2 steps	Synchronous/2 steps
	Required Computational Cost/Time for Encryption Key Generation		
	Low	High	Low
Security	Security Against Encryption Key Hijacking Attack		
	Vulnerable	Average	High
	Prevention of Key Generation Information Exposure		
	Prevention	Prevention	Prevention
	Randomness/Unpredictability of Encryption Key		
	Randomness /Unpredictability	Randomness /Unpredictability	Randomness /Unpredictability

3. Approach

This paper presents a method of encryption key generation between a sensor device that measures an environment and a gateway device used for collection. We propose a method of generating a dynamic encryption key using a block hash value that is dynamically generated based on the stored sensor data by referring to the methods of dynamically generating an unpredictable block hash by transaction history in the studies of [23,24]. The proposed method is illustrated in Figure 4.



Device$_{Sensor}$
SensorData
Device$_{gateway}$
SensorData
TransactionPool$_{Sensor}$
SensorData history
{SensorData$_1$,SensorData$_2$,SensorData$_3$ .... SensorData$_k$}
K: BlockCondition
TransactionPool(Hash)
Block$_{n-1}$
Block$_n$
Block$_{n-1.}$ (Hash)
Block$_n$ (Hash$_)$ = Hash(TransactionPool(Hash) +Block$_{n-1}$. (Hash))
Symmetric key - AES Key Generator (Seed) = DeviceKey$_N$
SensorData
TransactionPool$_{gateway}$
SensorData history
{SensorData$_1$,SensorData$_2$,SensorData$_3$ .... SensorData$_k$}
K: BlockCondition
TransactionPool(Hash)
Block$_{n-1}$
Block$_n$
Block$_{n-1.}$ (Hash)
Block$_n$ (Hash$_)$ = Hash(TransactionPool(Hash) +Block$_{n-1}$. (Hash))
Symmetric key - AES Key Generator (Seed)=DeviceKey$_n$

Figure 4. Concept of dynamic encryption key using block hash value.

The sensor device ($\text{Device}_{\text{sensor}}$) periodically sends environmental sensor values (sensor) to the gateway device ($\text{Device}_{\text{gateway}}$). The sensor device stores the send sensor values in the transaction pool (TransactionPool) and the gateway device also stores the received sensor values in its transaction pool. The send and received sensor data are stored in the transaction pool of each device. When a block generation condition (K transactions) is met, the hash value of N^{th} block is generated by hashing the values obtained by adding the hash value of a previous block to the hash value of current transaction pool in generating the subsequent N^{th} block. The hash value of N^{th} block is used as a seed value of symmetric key generation algorithm (AES) to generate a new N^{th} symmetric key. This encryption key is called DeviceKey_n, i.e., the N^{th} dynamic encryption key. When a dynamic encryption key is generated through this method, a seed value for unlimited number of dynamic encryption key can be generated for an environment sensor value that cannot be reproduced. In order to obtain the newly generated N^{th} dynamic block key information, it is necessary to know all the sensor data values sent and received, between devices. Additionally, it is impossible to know a key generated next even if the current N^{th} key is stolen key because all the histories of $N-1^{\text{th}}$ key must be obtained.

However, if this is applied to a low-power wireless communication environment as shown in Figure 5, encryption key mismatch occurs due to packet loss.

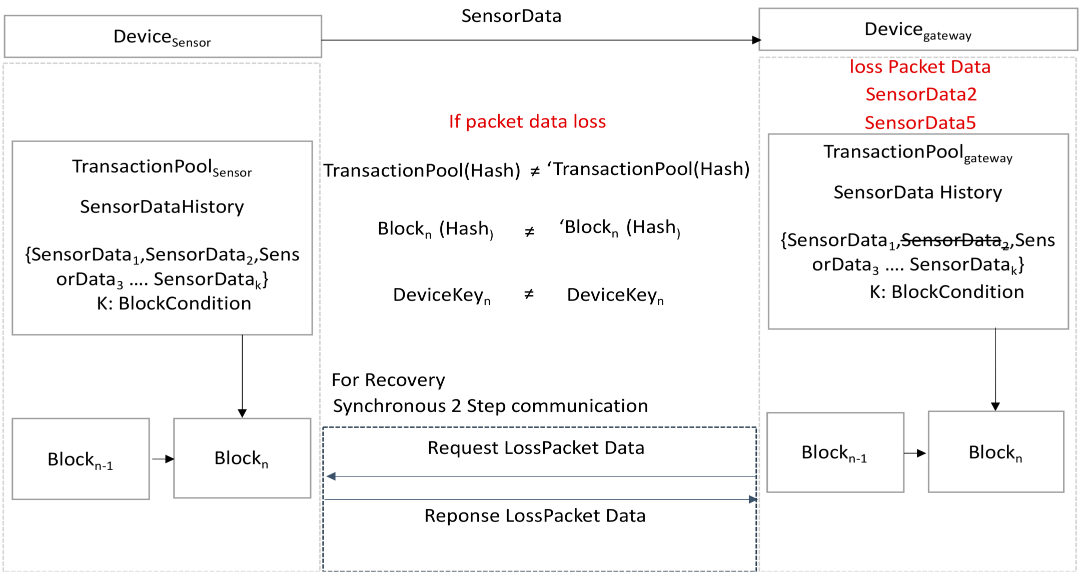
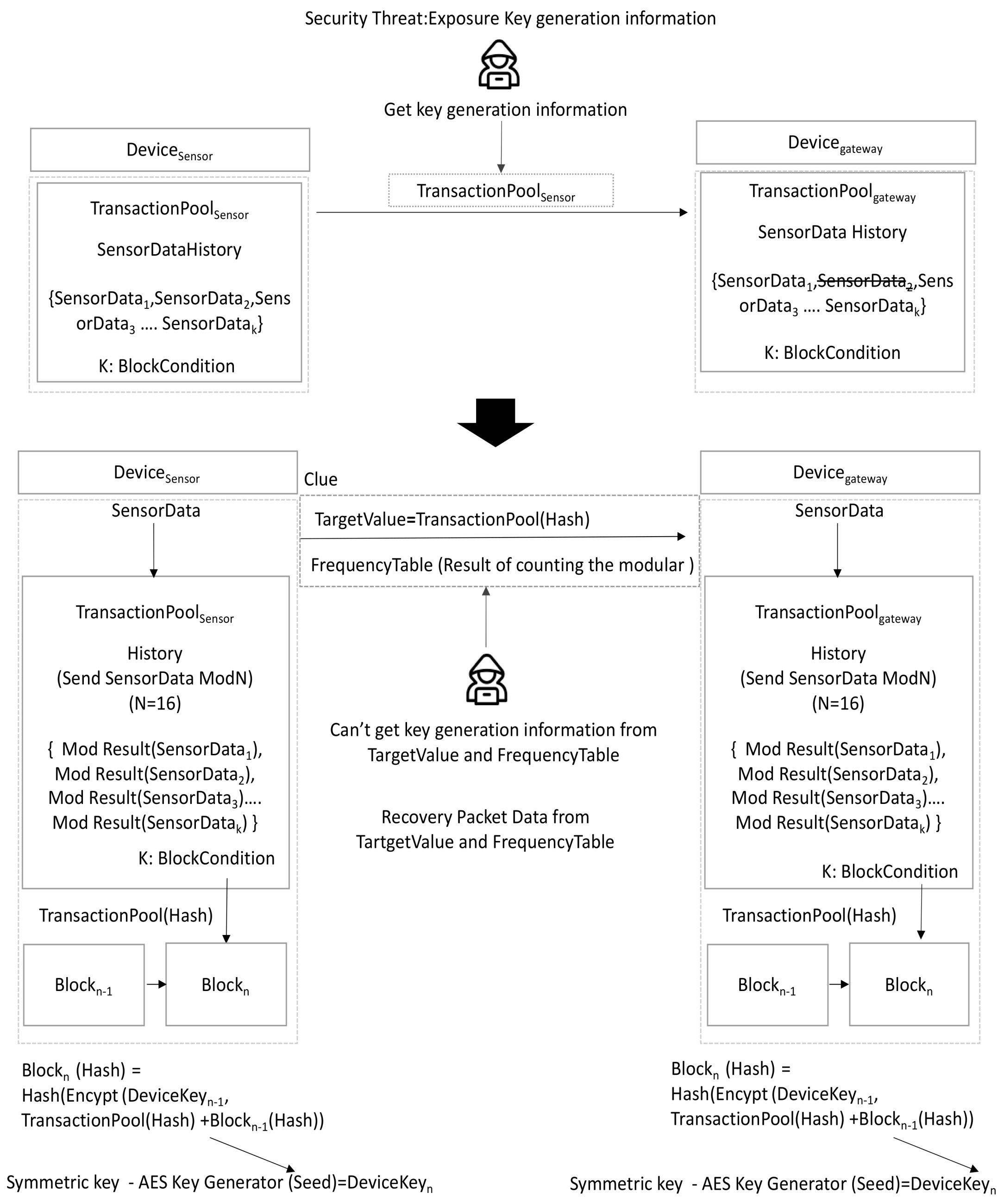

SensorData
$\text{Device}_{\text{Sensor}}$
$\text{Device}_{\text{gateway}}$
loss Packet Data
SensorData2
SensorData5
If packet data loss
$\text{TransactionPool}_{\text{Sensor}}$
SensorDataHistory
$\{\text{SensorData}_1, \text{SensorData}_2, \text{SensorData}_3 \ldots \text{SensorData}_k\}$
K: BlockCondition
TransactionPool(Hash) ≠ ‘TransactionPool(Hash)
Block_n (Hash) ≠ ‘Block_n (Hash)
DeviceKey_n ≠ DeviceKey_n
$\text{TransactionPool}_{\text{gateway}}$
SensorData History
$\{\text{SensorData}_1, \cancel{\text{SensorData}_2}, \text{SensorData}_3 \ldots \text{SensorData}_k\}$
K: BlockCondition
For Recovery
Synchronous 2 Step communication
Request LossPacket Data
Reponse LossPacket Data
Block_{n-1}
Block_n
Block_{n-1}
Block_n

Figure 5. Dynamic encryption key mismatch due to packet loss.

To address this issue, a method of generating the same dynamic encryption key through one-time asynchronous communication is proposed in consideration of packet loss. To generate the same dynamic encryption key with one-time asynchronously communication, there is a single option of transmitting all the transaction pool data from a sensor device whenever a new key is generated under the assumption that a message is lost by the gateway device.

However, as shown in Figure 6, the data stored in the transaction pool are encryption key generation information, and security threats may occur due to exposure during transmission. Accordingly, the solution is to deliver only a clue. A malicious attacker cannot know key generation information with only the clue, but a gateway device is capable of restoring a lost message with only the clue. This paper proposes a dynamic encryption key generation method as shown in the Figure 6 below regarding the clue idea.



Security Threat:Exposure Key generation information
Get key generation information
Device_{Sensor}
TransactionPool_{Sensor}
SensorDataHistory
{SensorData1,SensorData2,SensorData3 …. SensorDatak}
K: BlockCondition
TransactionPoolSensor
Devicegateway
TransactionPoolgateway
SensorData History
{SensorData1,~~SensorData2~~,SensorData3 …. SensorDatak}
K: BlockCondition
DeviceSensor
Clue
TargetValue=TransactionPool(Hash)
FrequencyTable (Result of counting the modular)
Devicegateway
SensorData
TransactionPoolSensor
History (Send SensorData ModN) (N=16)
{ Mod Result(SensorData1), Mod Result(SensorData2), Mod Result(SensorData3)…. Mod Result(SensorDatak) }
K: BlockCondition
Can't get key generation information from TargetValue and FrequencyTable
Recovery Packet Data from TartgetValue and FrequencyTable
TransactionPoolgateway
TransactionPool(Hash)
Blockn-1
Blockn
Blockn (Hash) = Hash(Encypt (DeviceKeyn-1, TransactionPool(Hash) +Blockn-1(Hash))
Symmetric key - AES Key Generator (Seed)=DeviceKeyn

Figure 6. Proposed dynamic encryption key method.

Even if a packet loss occurs, the sensor device and gateway device can generate the same transaction pools with one-time asynchronously communication, in turn the same Nth DeviceKey$_n$ is generated. For detailed explanation of the proposed method, its notations are described in Section 3.1 and the details of each proposed method are described in Section 3.2 System Model and Sections 3.3–3.6.

3.1. Notation

The notations used in the method proposed in this paper are shown in the Table 2 below.

Table 2. Notation.

Notation	Description
SensorData_i	*i*-th sensor data.
KeyGenerationInfo	Encryption key generation information.
$\text{Device}_{\text{Sensor}}$	Sensor device
$\text{Device}_{\text{gateway}}$	Gateway device that receives and collects sensor data
DataPeriodic	Data transmission cycle from sensor device to gateway device
$\text{ModResult}(\text{SensorData}_i, \text{N})$	Modular result of SensorData_i by N
DataCount_i	The counter value of the *i*-th data send by the sensorDevice
TransactionPool	Pool that stores the modular results of sensor
$\text{TransactionPool}(\text{DataCount}_i, \text{Modresult})$	Store {Key:DataCount_i Value:modular results} value in TransactionPool
TargetValue	$\text{TransactionPool}_{\text{sensor}}$ hashed through SHA256
Block_n	N^{th} block stored by sensor device and gateway device. Only previous hash value and current hash value exist in the block. The hash value of the block is the seed value of a new dynamic encryption key.
BlockCondition_K	Block generation condition. A block is generated when the number of modular results stored in the transaction pool satisfies k.
FrequencyTable	Frequency table indicating the frequency of modular result of the data
FrequencyCount(i,list)	Count value corresponding to i in the list items
FrequencyTable(i,result)	Generate FrequencyTable for {Key: i, value: result}
TransformFrequencyTable (Hash,FrequencyTable)	FrequencyTable transformation Based on hash value
Devicekey_n	N^{th} dynamic encryption key
lossFrequencyList	Loss Modular result list. The Difference between $\text{FrequencyTable}_{\text{sensor}}$ and $\text{FrequencyTable}_{\text{gateway}}$
PermutationList(lossFrequencyList)	List of possible permutations from lossFrequencyList
CandidateNonce	Candidate nonce for recovering loss packets on $\text{FrequencyTable}_{\text{gateway}}$
FindNonce	Trial and error for recovering loss packets on $\text{FrequencyTable}_{\text{gateway}}$
Synchronization	Recover loss packets on the $\text{FrequencyTable}_{\text{gateway}}$ from CandidateNonce
Seed_n	Seed value of DeviceKey_n
Encrypt(Key,data)	Data encryption using Key
Decrypt(Key,data)	Data decryption using Key
initialize()	Initialization to generate N + 1 block after N^{th} block is generated. TransactionPool Initialization. DataCount_i Initialization

3.2. System Model

The details of the system model, where the scheme proposed in this paper is applied, are as follows:

- Communication between devices that generate dynamic encryption keys occurs between a sensor device and a gateway device.
- The sensor device and gateway device use Bluetooth low-energy, which can encounter communication delay and message loss due to signal interference between devices.
- The sensor device and gateway device use an asynchronous network control flow.
- A sensor device measures environment data that cannot be reproduced.
- In ModResult (SensorData, N), N is set to 16.

- A sensor device transmits a measured environment sensor data value to a gateway device with DataPeriodic = 1 (1 s interval).
- The BlockCondition$_K$ is 64. When 64 data entries are filled in a transaction pool, block generation is attempted. The reason behind setting the number to 64 is because a malicious attacker must randomly substitute 2^{256} number of cases to guess the transaction pool, which is the seed value of an encryption key. If DataPeriodic is 1, a new encryption key is generated every 64 s, but it is impossible to guess the key through random substitution of 2^{256} cases within 64 s.
- The initial connection and registration between devices are registered by a trusted administrator.
- The block structure stores only the hash values of previous block and current block by considering low-performance devices.
- As for the blocks, only the current block and previous block are stored in consideration of low-performance devices. Unlike cryptocurrency transaction, sensor data do not have a structure of verifying current data from previous data, thus it is not necessary to have all the history.
- Sections 3.3–3.6 explain the descriptions on the process of generating DeviceKey$_n$ (Nth key).

3.3. Storing Transaction Pool

The sensor device (Device$_{Sensor}$) encrypts the periodically measured environment sensor data (SensorData$_i$) into DeviceKey$_{n-1}$ and transmits them to the registered gateway device (Device$_{gateway}$). The transmitted SensorData$_i$ contain environment sensor values and DataCount for identification. Whenever the sensor device transmits data, it executes ModResult(SensorData$_i$,16) and stores the result in TransactionPool$_{Sensor}$ along with the counter number. The result of ModResult is one of the values from 0 to F. Until BlockCondition$_{64}$ is satisfied, each ModResult of SensorData$_1$, SensorData$_2$, SensorData$_3$, ..., SensorData$_{64}$ is saved into TransactionPool. The TransactionPool accumulates up to 64 values from 0 to F, which are the modular results. The Device$_{gateway}$ also executes ModResult (SensorData, 16) for received data and stores the result in TransactionPool$_{gateway}$ along with the counter number.

If there is no packet loss, TransactionPool$_{gateway}$ and TransactionPool$_{Sensor}$ should have the same transaction pool. On the other hand, if there is a lost packet, the TransactionPool$_{gateway}$ shows the lost data. Figure 7 shows the mechanism of storing Transaction Pool. The pseudo code is as Algorithm 1.

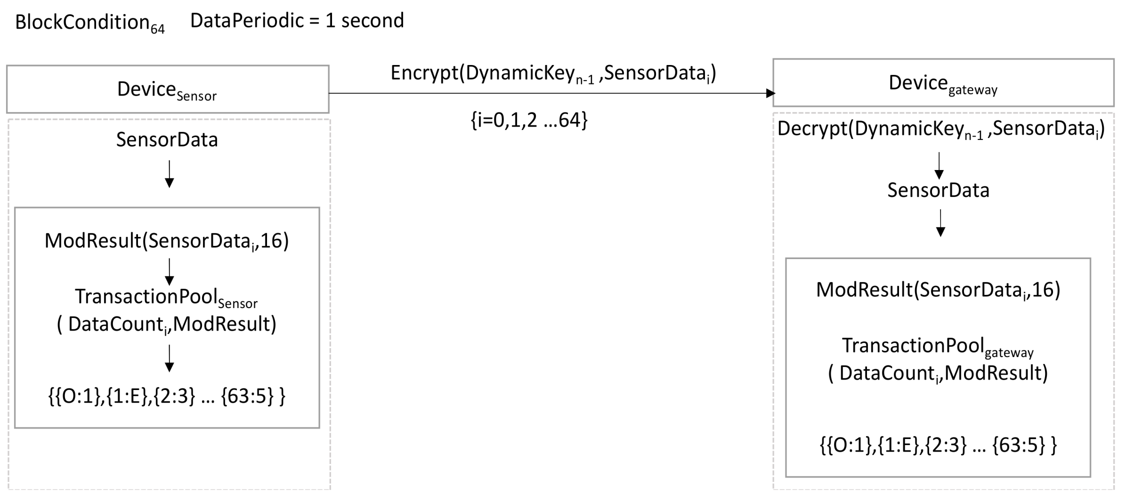

BlockCondition$_{64}$ DataPeriodic = 1 second
Device$_{Sensor}$
Encrypt(DynamicKey$_{n-1}$,SensorData$_i$)
{i=0,1,2 ...64}
Device$_{gateway}$
SensorData
ModResult(SensorData$_i$,16)
TransactionPool$_{Sensor}$ (DataCount$_i$,ModResult)
{{O:1},{1:E},{2:3} ... {63:5} }
Decrypt(DynamicKey$_{n-1}$,SensorData$_i$)
SensorData
ModResult(SensorData$_i$,16)
TransactionPool$_{gateway}$ (DataCount$_i$,ModResult)
{{O:1},{1:E},{2:3} ... {63:5} }

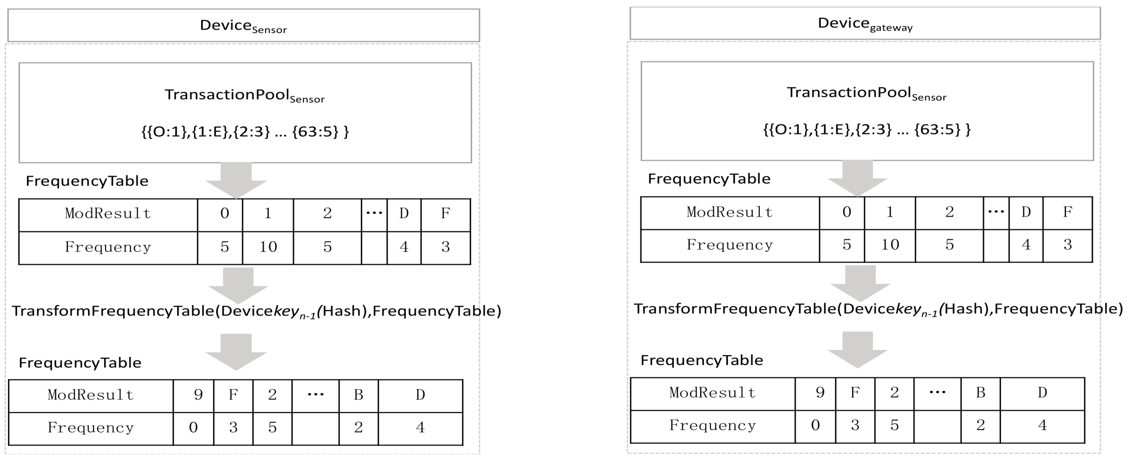
Figure 7. Transaction pools stored by sensor device and gateway device.

Algorithm 1. TransactionPool

```
1: procedure TransactionPool
2:   Device$_{Sensor}$ executes:
3:     while BlockCondition$_{64}$ do
4:       Send Encrypt (DeviceKey$_{n-1}$,sensorData$_i$) to Device$_{gateway}$
5:       ModResult (DataCount$_i$, 16)-> ModResult$_i$
6:       TransactionPool (Count, ModResult$_i$)
7:   Device$_{gateway}$ executes:
8:     while BlockCondition$_{64}$ do
9:       Receive Encrypt (DeviceKey$_{n-1}$,sensorData$_i$) from Device$_{gateway}$
10:        Decrypt (DeviceKey$_{n-1}$,sensorData$_i$)
11:        ModResult (sensorData$_i$, 16)-> ModResult$_i$
12:        TransactionPool (DataCount$_i$, ModResult$_i$)
13: end procedure
```

3.4. *FrequencyTable Generation*

If there is lost data, more than 2 steps of message exchange are required in a synchronous communication to generate the same encryption key between devices. A method is required in which the Device$_{gateway}$ requests the lost data and Devices$_{ensor}$ transmits the lost data as a response. However, this cannot be applied in an asynchronous low-power wireless communication environment. To solve this problem with at least one-time asynchronous communication, there is a method of transmitting TransactionPool$_{Sensor}$ when generating a key by considering a message loss in advance in the Device$_{sensor}$. However, if TransactionPool$_{Sensor}$ is stolen maliciously, the generated information can be exposed. To prevent this, a frequency table is used. Figure 8 shows frequency table generation. The frequency table is the result of counting the modular operation results from 0 to F of each transaction pool. The Device$_{sensor}$ generate the following frequency table (Frequency>Table$_{sensor}$) by obtaining the frequency of TransactionPool$_{Sensor}$ where the modular result of transmitted data is stored.



Device$_{Sensor}$
TransactionPool$_{Sensor}$
{{O:1},{1:E},{2:3} … {63:5} }
FrequencyTable
TransformFrequencyTable(Device*key$_{n-1}$*(Hash),FrequencyTable)
FrequencyTable
Device$_{gateway}$
TransactionPool$_{Sensor}$
{{O:1},{1:E},{2:3} … {63:5} }
FrequencyTable
TransformFrequencyTable(Device*key$_{n-1}$*(Hash),FrequencyTable)
FrequencyTable

ModResult	0	1	2	…	D	F
Frequency	5	10	5		4	3

ModResult	9	F	2	…	B	D
Frequency	0	3	5		2	4

ModResult	0	1	2	…	D	F
Frequency	5	10	5		4	3

ModResult	9	F	2	…	B	D
Frequency	0	3	5		2	4

Figure 8. FrequencyTable.

The Device$_{gateway}$ also creates FrequencyTable$_{gateway}$ from TransactionPool$_{gateway}$. If there was no data loss, FrequencyTable$_{sensor}$ and FrequencyTable$_{gateway}$ would be the same. If there were two lost data entries, i.e., data having ModResult of 2 and 3, the frequency of 2 and 3 of FrequencyTable$_{gateway}$ would be stored less by 1 each. Additionally, the sum of frequencies will be $64 - 2 = 62$. If the FrequencyTable created in Device$_{sensor}$ is transmitted as it is, a malicious attacker can know the frequency of modular result in the transaction pool, thus the order is changed. To change the order, the result place of frequency is changed by using the hash value of Block$_{n-1}$. They are changed in the order of hash values, and further, padding is performed in order if there is no ModResult. If the hash value of Block$_{\text{n-1}}$ is 9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08, the order of frequency table is changed in the order of 9 f 8 6 d 0 1 4 c 7 d 5 a 2 e 3 b d. If a malicious attacker intends to obtain the original FrequencyTable by stealing a frequency table, 16! = 20,922,789,888,000 number of cases should be computed for the acquisition. Because the Device$_{\text{gateway}}$ has the same hash value of Block$_{\text{n-1,}}$ its order is changed in the same way as the frequency table. The pseudo code is as Algorithm 2.

Algorithm 2. FrequencyTable

```
1: procedure FrequencyTable
2:  Device_Sensor executes:
3:      list = transactionPool
4:        while i < F do:
5:            FrequencyCount(i,list) -> result
6:            FrequencyTable(i, result)
7:            increment i
8:        TransformFrequencyTable(Devicekey_n-1(Hash),FrequencyTable) ->
          FrequencyTable_sensor
9:  Device_gateway executes:
10:     list = transactionPool
11:       while i < F do:
12:           FrequencyCount(i,list) -> result
13:           FrequencyTable(i,result)
14:           increment i
15:       TransformFrequencyTable(Devicekey_n-1(Hash),FrequencyTable) ->
          FrequencyTable_gateway
16: end procedure
```

3.5. Transmission of Target Value and FrequencyTable, Synchronization of Transaction Pool

If the Device$_{\text{sensor}}$ transmits FrequencyTable$_{\text{sensor}}$, the Device$_{\text{gateway}}$ can know which frequency (of ModResult) is missing by comparing with FrequencyTable$_{\text{gateway}}$. However, only the number of cases can be found for the position of TransactionPool only with ModResult. To generate the same TransactionPool by using only the frequency difference, the answer sheet of entire transaction pool is required. However, if the answer sheet is transmitted, a malicious attacker can steal it and know the transaction pool, i.e., the key generation information. To prevent this, a hash function is used. Due to its preimage resistance, it is almost impossible to find out an input value by looking at the hash result. Figure 9 shows how to synchronization of transactionpool. The Device$_{\text{sensor}}$ calculated the hash value of TransactionPool and sets it as TargetValue. When BlockCondition$_{64}$ is met, the Device$_{\text{sensor}}$ sends a message including TargetValue and FrequencyTable$_{\text{sensor}}$ to notify the generation of a new dynamic key. Even if a malicious attacker steals TargetValue and FrequencyTable$_{\text{sensor}}$, it is impossible to find out the original transaction pool within 64 s which is the generation cycle of the subsequent encryption key.

The Device$_{\text{gateway}}$ that received TargetValue$_{\text{sensor}}$ and FrequencyTable$_{\text{sensor}}$ calculates TargetValue when there is no lost message, compares it with the received TargetValue, and generates a new dynamic encryption key in Section 3.6. If there is data loss, it calculates ModResult by using the difference between FrequencyTable$_{\text{sensor}}$ and the lost FrequencyTable$_{\text{gateway}}$, and generates the same TransactionPool from the TargetValue. It calculates permutations by using the lost ModResult, substitutes the permutation results into the lost part of TransactionPool$_{\text{Gateway}}$, and attempts to obtain the same TargetValue through hashing based on trial and error. For example, suppose that 5 messages are lost, the lost modular results are {1:1 2:2 A:2}, and the index order of each lost message is 20, 21, 30, 44, 57. For the places of 20, 21, 30, 44, 57, there are 30 possible cases, i.e., $5!/(2! \times 2!)$ which are all permutations. For these 30 cases, attempts are made to find the case having the same value as the received TargetValue by filling in the empty index position and hashing the possible transaction pool. The case of having same value as the TargetValue indicates that the result is the same as $\text{TransactionPool}_{\text{sensor}}$. Through this process, the $\text{Device}_{\text{gateway}}$ will have the same TransactionPool as the $\text{Device}_{\text{sensor}}$ despite the data loss. Meanwhile, suppose that a malicious attacker steals TargetValue and $\text{FrequencyTable}_{\text{sensor}}$. If the frequency of ModResult is the same, the attacker should find out the TargetValue through repetitive hashing for 16! × (64!)/(4! ×16) number of cases. It is impossible to find out the TargetValue within the $\text{BlockCondition}_{64}$, i.e., 64 s which is the generation cycle of the subsequent encryption key. The pseudo code is as Algorithm 3.

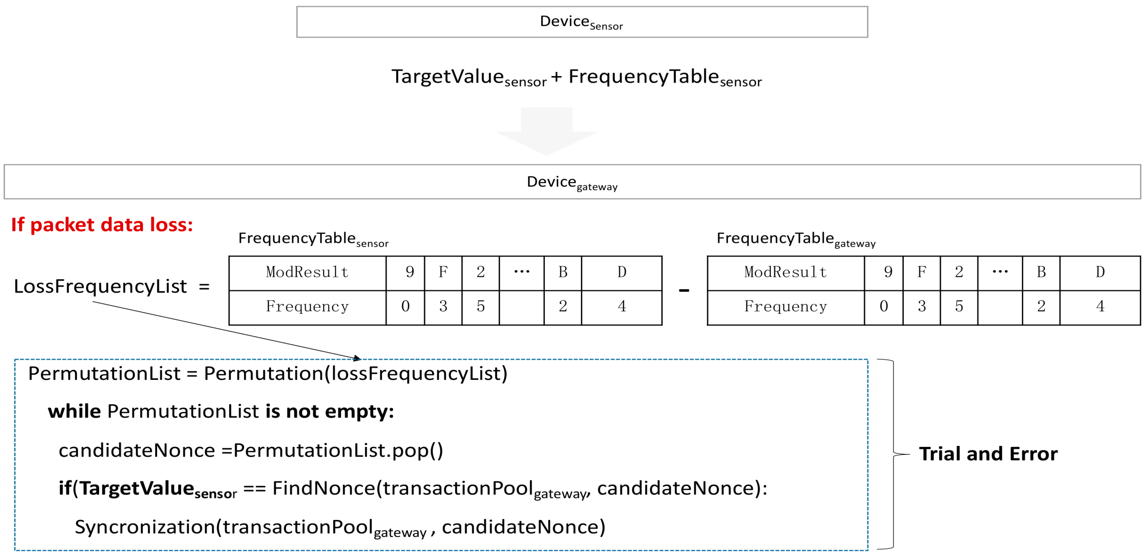

Device$_{\text{Sensor}}$
TargetValue$_{\text{sensor}}$ + FrequencyTable$_{\text{sensor}}$
Device$_{\text{gateway}}$
If packet data loss:
LossFrequencyList =
FrequencyTable$_{\text{sensor}}$
FrequencyTable$_{\text{gateway}}$
ModResult 9 F 2 ⋯ B D
Frequency 0 3 5 2 4
–
ModResult 9 F 2 ⋯ B D
Frequency 0 3 5 2 4
PermutationList = Permutation(lossFrequencyList)
while PermutationList is not empty:
candidateNonce =PermutationList.pop()
if(TargetValue$_{\text{sensor}}$ == FindNonce(transactionPool$_{\text{gateway}}$, candidateNonce):
Syncronization(transactionPool$_{\text{gateway}}$, candidateNonce)
Trial and Error

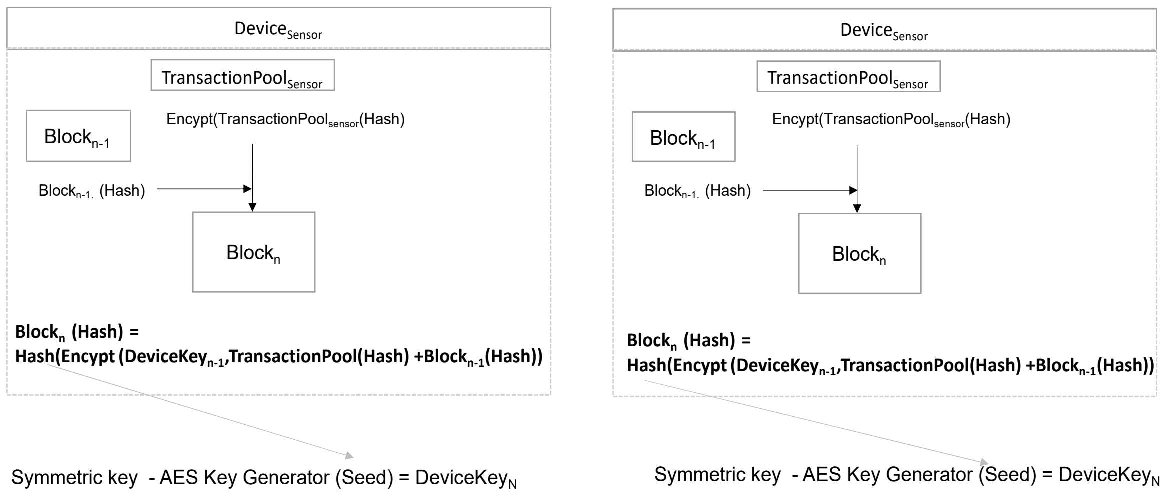
Figure 9. TransactionPool Synchronization.

Algorithm 3. TransactionPool Synchronization

```
1:  procedure TransactionPool Synchronization
2:       $Device_{Sensor}$ executes:
3:           $TargetValue_{sensor}$ = SHA256(list)
4:           Send Encrypt($DeviceKey_{n-1}$,$TargetValue_{sensor}$ + $FrequencyTable_{sensor}$) to
             $Device_{gateway}$
5:           Call Module DeviceKeyGeneration()
6:    $Device_{gateway}$ executes:
7:               $TargetValue_{gateway}$ = SHA256(list)
8:               Receive Encrypt ($DeviceKey_{n-1}$,$TargetValue_{sensor}$+ $FrequencyTable_{sensor}$) from
9:           $Device_{sensor}$
10:        Decrypt($DeviceKey_{n-1}$, $TargetValue_{sensor}$+ $FrequencyTable_{sensor}$)
12:    if loss data then:
13:        lossFrequencyList = $FrequencyTable_{sensor}$- $FrequencyTable_{gateway}$
14:        PermutationList = Permutation(lossFrequencyList)
17:        while PermutationList is not empty:
18:            candidateNonce = PermutationList.pop()
19:            if($TargetValue_{sensor}$ == FindNonce(transactionPool, candidateNonce):
20:            Syncronization(transactionPool, candidateNonce)
21:            Call Module DeviceKeyGeneration()
22:    else:
23:        if($TargetValue_{sensor}$ == $TargetValue_{gateway}$):
24:            Call Module DeviceKeyGeneration()
25: end procedure
```

3.6. Generation Dynamic Encryption Key

The $\text{Device}_{\text{sensor}}$ and $\text{Device}_{\text{gateway}}$ that have the same TransactionPool independently generate the Nth dynamic encryption key. Figure 10 shows how to generate dynamic encryption key. The TransactionPool hash value is encrypted with Encrypt (DeviceKey_{n-1},TransactionPool(Hash)). The reason for encryption is to prevent exposure as the TransactionPool (Hash) value has already been sent TargetValue. The $\text{Block}_{\text{n-1}}$ hash value is added to this encrypted value and the result is set to $\text{BlockInfo}_{\text{N}}$. One more hashing of $\text{BlockInfo}_{\text{N}}$ becomes the hash value of Nth block.



Device_{Sensor}
TransactionPool_{Sensor}
Encypt(TransactionPool_{sensor}(Hash)
Block_{n-1}
Block_{n-1.} (Hash)
Block_n
Block_n (Hash) =
Hash(Encypt (DeviceKey_{n-1},TransactionPool(Hash) +Block_{n-1}(Hash))
Symmetric key - AES Key Generator (Seed) = DeviceKey_N

Figure 10. Dynamic encryption key generation.

The hash value of Block_{n} is used as a seed value to generate the key for the new symmetric key AES. This newly generated encryption key is the Nth $\text{DeviceKey}_{\text{n}}$. Both $\text{Device}_{\text{sensor}}$ and $\text{Device}_{\text{gateway}}$ will generate the same symmetric encryption key. The pseudo code is as Algorithm 4.

Algorithm 4. DeviceKeyGeneration

```
1: procedure DeviceKeyGeneration
2:     $\text{Device}_{\text{Sensor}}$ executes:
3:         ($\text{Block}_{\text{n-1}}$ (Hash)+Encrypt($\text{DeviceKey}_{\text{n-1}}$,TransactionPool(Hash)) -> $\text{Blockinfo}_{\text{n}}$
4:         $\text{Seed}_{\text{n}}$ = SHA256($\text{Blockinfo}_{\text{n}}$)
5:         Call module AESKeyGenerator($\text{Seed}_{\text{n}}$) -> $\text{DeviceKey}_{\text{n}}$
6:     $\text{Device}_{\text{gateway}}$ executes:
7.         ($\text{Block}_{\text{n-1}}$ (Hash)+Encrypt($\text{DeviceKey}_{\text{n-1}}$,TransactionPool(Hash)) -> $\text{Blockinfo}_{\text{n}}$
8:         $\text{Seed}_{\text{n}}$ = SHA256($\text{Blockinfo}_{\text{n}}$)
9:         Call module AESKeyGenerator($\text{Seed}_{\text{n}}$) -> $\text{DeviceKey}_{\text{n}}$
10: end procedure
```

4. Experiment Verification

The proposed method in this paper is tested and verified in terms of availability and security in a low-power wireless communication IoT environment. Detailed items are as follows:

Availability aspect

- Communication method/step for encryption key generation (theoretical verification).
- Communication method/step for encryption key recovery (theoretical verification).
- Key generation time per packet loss (experimental verification).
- Key generation time, Memory used, CPU used for key generation operation (experimental verification).

Security aspect

- Randomness and unpredictability of encryption key (theoretical verification).
- Prevention of key generation information exposure (experimental verification).
- Security against Encryption Key Hijacking Attack (theoretical verification).

4.1. Configuration of Experimental Environment

To perform an experiment for the method proposed in this paper, conducted the experiments in the Device$_{\text{sensor}}$ and the Device$_{\text{gateway}}$ of the Raspberry PiB+ model. The package configuration is as Tables 3 and 4.

Table 3. System Configuration.

Category	Description
CPU	Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64bit SoC 1.4GHz
RAM	1GB LPDDR2 SDRAM
OS	RaspbianStrech
Language	Python 3.6
Communication	Bluetooth low energy 4.2
External Library	-itertools: permutation -hashlin: sha256 -cryptography, cryptodome: encryption module

Table 4. Software Package.

Package	Description
communication	sender: Bluetooth connection and data send receiver: Bluetooth connection and data receive
encryption module	encryption module libraries
algorithm module	frequencytable: frequencytable manage transactionPool: transactionPool manage findnonce: trial and error manage permuatioan: permutation manage
launcher	sensor: execute Device$_{\text{sensor}}$ device: execute Device$_{\text{Gateway}}$
test	KeyGeneration_Time: key generation time experiment KeyGneration_Time_packetDataloss: key generation time experiment with packet loss Memory _Check: memory used experiment Permutation_Check: permutation experiment
logging	log data KeyGeneration_Time.log KeyGneration_Time_packetDataloss.log Memory _Check.log

4.2. Availability Aspect

4.2.1. Communication Method/Step for Encryption Key Generation

Theory

Device$_{\text{Sensor}}$ and Device$_{\text{Gateway}}$ generate the same encryption key through asynchronous one-step communication.

Proof

(1) In the event of packet loss, $\text{Device}_{\text{Sensor}}$ and $\text{Device}_{\text{Gateway}}$ are asynchronous or synchronous if there is a request/response for generate $\text{DeviceKey}_{\text{n}}$
(2) Communication Steps for $\text{Device}_{\text{Sensor}}$ and $\text{Device}_{\text{Gateway}}$ to generate $\text{DeviceKey}_{\text{n}}$
(3) $\text{Device}_{\text{sensor}}$ and the $\text{Device}_{\text{gateway}}$ generates the same encryption key as (1) (2)

(1) in case of; $\text{Device}_{\text{sensor}}$ sends $\text{FrequencyTable}_{\text{sensor}}$ and $\text{TargetValue}_{\text{sensor}}$ to key generation notification if $\text{BlockCondition}_{\text{K}}$ is satisfied. $\text{Device}_{\text{gateway}}$ that receives the message generates $\text{DeviceKey}_{\text{n}}$. It is an asynchronous method because there is no response to the generate encryption key. (2) There is only one step communication to generate a key from the $\text{Device}_{\text{Sensor}}$. (3) $\text{Device}_{\text{sensor}}$ and $\text{Device}_{\text{gateway}}$ generate the same encryption key based on the same sensor data history. In conclusion, $\text{Device}_{\text{sensor}}$ and $\text{Device}_{\text{gateway}}$ generate the same encryption key through asynchronous 1 step communication.

4.2.2. Communication Method/Step for Encryption Key Recovery

Theory

In the event of packet loss, $\text{Device}_{\text{Sensor}}$ and $\text{Device}_{\text{Gateway}}$ generate the same encryption key through asynchronous 1 step communication.

Proof

(1) In the event of packet loss, $\text{Device}_{\text{Sensor}}$ and $\text{Device}_{\text{Gateway}}$ are asynchronous or synchronous if there is a request/response for recovery.
(2) Communication Steps for $\text{Device}_{\text{Sensor}}$ and $\text{Device}_{\text{Gateway}}$ to recovery in the event of packet loss.
(3) $\text{Device}_{\text{sensor}}$ and the $\text{Device}_{\text{gateway}}$ generates the same encryption key as (1) (2).

(1) In case of; $\text{Device}_{\text{sensor}}$ sends $\text{FrequencyTable}_{\text{sensor}}$ and $\text{TargetValue}_{\text{sensor}}$ to key generation notification if $\text{BlockCondition}_{\text{K}}$ is satisfied considering packet loss. It is an asynchronous method because there is no response to the loss of packets from $\text{Device}_{\text{Gateway.}}$ $\text{Device}_{\text{gateway}}$ generates the same encryption key from $\text{FrequencyTable}_{\text{sensor}}$, $\text{TargetValue}_{\text{sensor}}$ even if loss packet. (2),(3) In conclusion, $\text{Device}_{\text{sensor}}$ and $\text{Device}_{\text{gateway}}$ generate the same encryption key through asynchronous 1 step communication for recovery.

4.2.3. Key Generation Time per Packet Loss

In the proposed method, the key generation time varies depending on packet loss. If it takes a long time to generate a key when there is a packet loss, the method is not suitable because it causes a delay in the system. In this section, the key generation time depending on packet loss is measured. Each execution measures the average time for 1000 times of key generation depending on packet loss to examine the availability of the proposed method. The formula for measuring performance is as Equation (1).

$$\mu\left(\text{T}^{\text{l}}_{\text{keygen}}\right) = \sum_{\text{i}=0}^{\text{n}} \frac{\text{T}^{\text{end}}_{\text{key}_\text{i}} - \text{T}^{\text{start}}_{\text{key}_\text{i}}}{\text{n}} \tag{1}$$

$$\mu\left(\text{T}^{\text{l}}_{\text{keygen}}\right)\ \text{:l loss packet , key generation average time.}$$

$$\text{T}^{\text{start}}_{\text{key}_\text{i}}:\ \text{i'th key generation start point}$$

$$\text{T}^{\text{end}}_{\text{key}_\text{i}}:\ \text{i'th key generation end point}$$

As shown Figure 11, proposed method allows key generation without system delay of up to nine packet losses in a DataPeriodic:1. When $\text{BlockCondition}_{\text{K}}$ was set to 64 (defend a malicious attacker's transaction pool brute force attacks), encryption keys can be generated without delay even if 14.06% of packet loss. When considering packet data loss rate of 2% due to interference between devices in Bluetooth low energy communication. The method proposed in the paper are applicable.

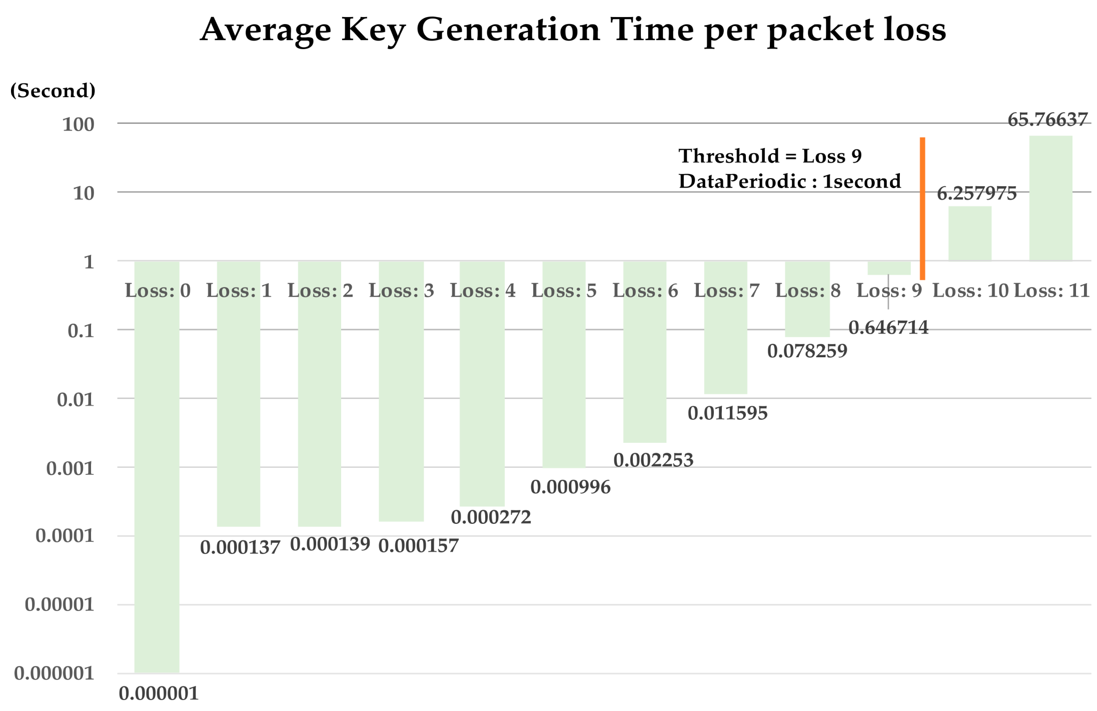


Figure 11. Experiment result of Section 4.2.3.

4.2.4. Key Generation Time, Memory Used, CPU Used

The methods proposed in the paper and the related work, OTP (SKEY), Diffe–Hellman, and Blockey, measure key generation time and memory used to compare availability. The average key generation time is measured by performing each algorithm 1000 times. The formula for measuring performance is as Equation (2).

$$\mu\left(T_{\text{keyGen}}\right) = \sum_{i=0}^{n} \frac{T_{\text{key}_i}^{\text{end}} - T_{\text{key}_i}^{\text{start}}}{n} \tag{2}$$

$$T_{\text{key}_i}^{\text{start}} \text{ i'th key generation start point.} \tag{3}$$

$$T_{\text{key}_i}^{\text{end}} \text{ i'th key generation end point.} \tag{4}$$

The average key generation time for each algorithm is shown in the Figure 12. In case of OTP, the key generation average time is set to zero because the key is generated in initial step. In the case of Diffe–Hellman, the key is generated using an asymmetric key, so it has the slowest key generation speed of 0.04211878 s. The method proposed and Blockey showed fast key generation speed results with 0.00000138 s and 0.00000148 s. Considering the loss rate of 2% of Bluetooth low energy communication, it was shown to be about 0.000137 s for 1–2 packet loss. The proposed method has shown that encryption keys can be generated and used without delay in the event of packet loss in a low-power wireless environment.

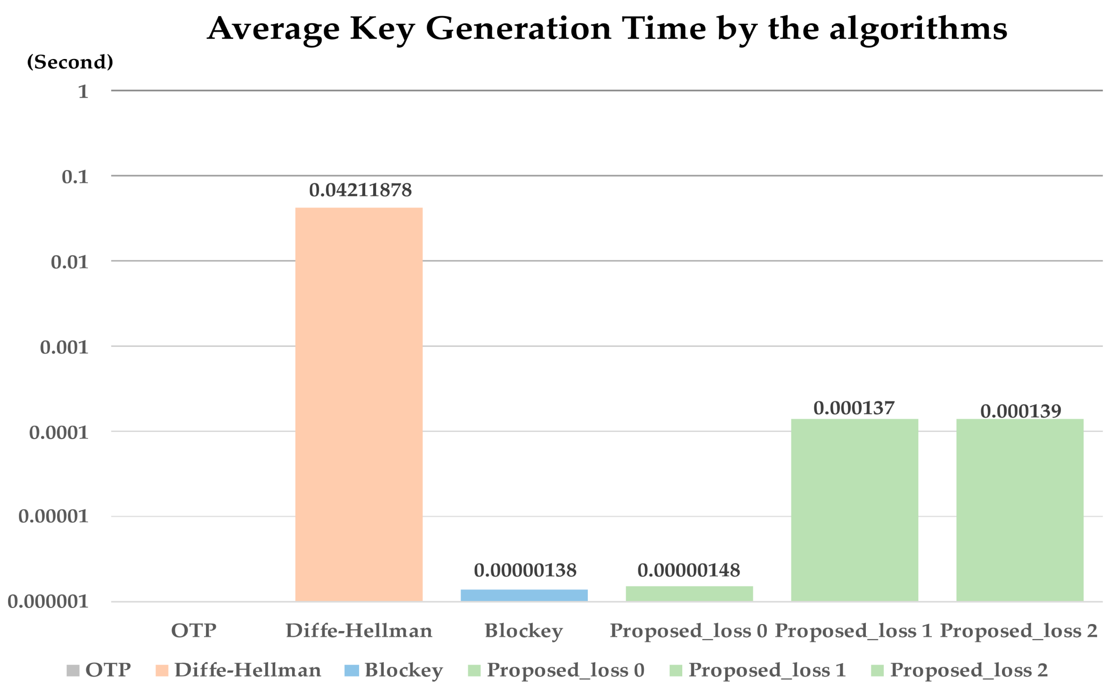


Figure 12. Experiment result of Section 4.2.4.

To measure the memory used, perform each algorithm 1000 times to measure the usage of key generation memory. Memory used measured the amount of memory used by the process when generating the key. Figure 13 is the result of an experiment. As shown Figure 13, in the case of Diffe–Hellman, the memory usage was the highest at 343.812 Kbyte because it generates asymmetric keys. Blockey and proposed method showed lower memory usage than Diffe–Hellman at 307.578 KB and 309.574 KB, respectively.

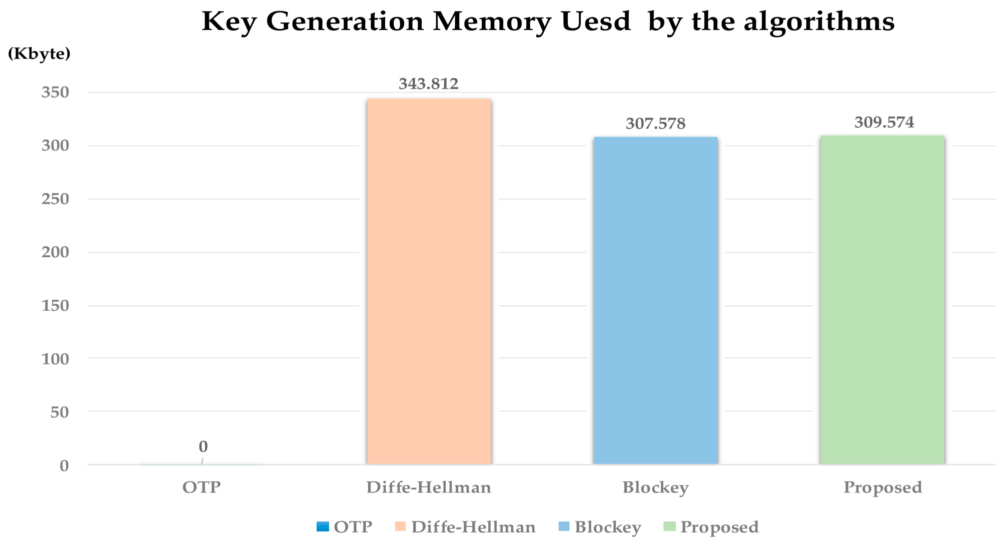


Figure 13. Experiment result of Section 4.2.4.

To measure the CPU used, perform each algorithm 1000 times to measure the usage of key generation CPU. CPU used measured the percentage of CPU used by the process when generating the key. Figure 14 is the result of an experiment. As shown Figure 14, in the case of Diffe–Hellman, the CPU usage was the highest at 11.4% because it generates asymmetric keys. Blockey and proposed method showed lower CPU usage than Diffe–Hellman at 1.7% and 1.8%, respectively.

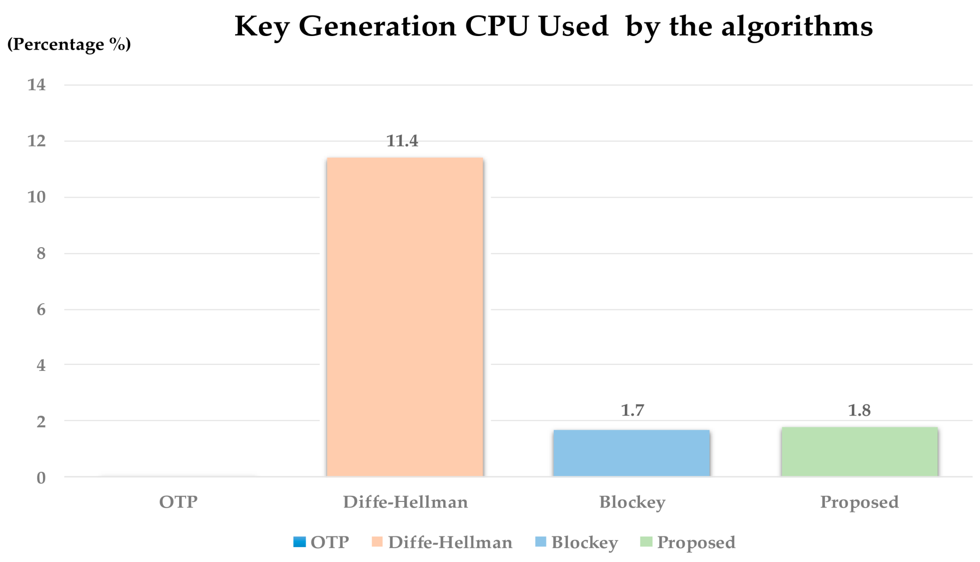


Figure 14. Experiment result of Section 4.2.4.

4.3. Security Aspect

4.3.1. Randomness and Unpredictability of Encryption Key

Dynamic encryption key must satisfy randomness and unpredictability. The method proposed in this paper is to generate hash values based on sensor data history that measure the environment and use them as seed values for generate encryption keys. This method satisfies randomness because it generates different hash values in an environment where measured sensor data change even slightly. It also satisfies unpredictability because it requires knowing or reproducing the history of all sensor data in $\text{Device}_{\text{Sensor}}$ and $\text{Device}_{\text{Gateway}}$ to know the next encryption key generated.

4.3.2. Prevention of key Generation Information Exposure

Key generation information should not be known even when a malicious hijacking attack. For this purpose, TargetValue and FrequencyTable were used. The experiment below measured the time of repetitive operation of the hash with increasing permutation. Detailed environment are as shown in the Table 5 below.

Table 5. System Configuration.

Category	Description
CPU	Intel Core i9 8Core 2.3Ghz
RAM	16GB 2400 MHZ DDR4
OS	Mac OS
Language	Python 3.6
External lib	-itertools: permutation

As shown Figure 15, the experimental result took approximately 16,137 s (approximately 4.5 h) on a 13! basis when experimenting with above-average computer performance. In $\text{BlockCondition}_{64}$, malicious attacker must generate a TransactionPool from TargetValue and FrequencyTable within 64 s. Malicious attacker should find out the TargetValue through repetitive hashing for $16! \times (64!)/(4! \times 16)$ number of cases. It is impossible to find out the TargetValue within the $\text{BlockCondition}_{64}$, i.e., 64 s. the method proposed in the paper prevent key generation information exposure attacks.

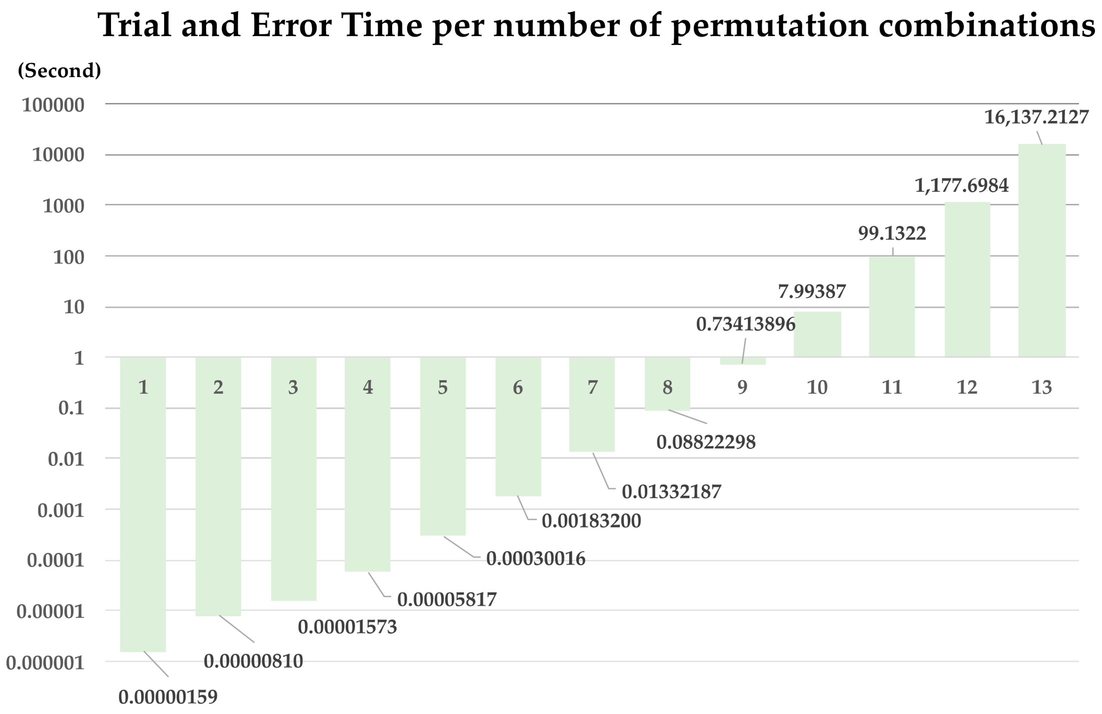


Figure 15. Experiment result of Section 4.3.2.

4.3.3. Security against Encryption Key Hijacking Attack

Suppose the proposed DeviceKey$_{n-1}$ key is stolen. Malicious attacker needs to know the value of the Block$_{n\text{-}1}$ hash to generate the Nth generated DeviceKey$_{n.}$ This requires a history of all sensor data sent and received to know the hash value. Alternatively, in order to know the hash value of n−1 at BlockCondition$_K$ = 64, malicious attacker make brute force attacks by the number of times the $2^{256} \times 2^{256} \times 2^{256} \ldots\ldots \times 2^{256} \times 2^{256}$ (n−1 times).

5. Conclusions

The Internet of Things' security threat extends to real physical systems, and economic damage and human life can be threatened. For the security of the Internet of Things, encryption key technology to identify and authentication trusted devices is important. Dynamic encryption keys, which are more secure than static encryption keys, can make them more secure in the Internet of Things. However, most of the Internet of Things low-power wireless communication uses a lot of asynchronous communication; thus, the traditional dynamic encryption keys based on synchronous are difficult to apply. There is also a dynamic encryption key synchronization problem due packet loss. In order to apply dynamic encryption keys in a low-power wireless environment, the problem of encryption key synchronization in asynchronous communication must be solved and also satisfying security. Table 6 is the result of a comparison of the related work and proposed method.

In this paper, the dynamic encryption key method applied with the mechanism of the blockchain was proposed to solve these problems. Based on the sensor data history between devices, hash value is generated dynamically, and a new dynamic encryption key is generated using the encryption key seed value. Moreover, the proposed method use the "clues (TargetValue and FrequencyTable)" to prevent key generation information exposure and to generate the same dynamic cryptographic key in an asynchronous/1step communication. The proposed method generates the same encryption key between devices, with only one step of asynchronous communication considering packet loss. Therefore, proposed method is asynchronous/1 step communication in both "encryption key generation communication method/steps" and "the encryption key recovery communication method/step". The key generation memory used, and the average key generation time were 0.00000148 s 307.578

Kbyte, respectively. It showed a lower level equivalent to the blockey for "computational cost/time for encryption key Generation". In comparison with the related work, the proposed method satisfied the aspects of availability in the low-power wireless communication environment. The "security against encryption key hijacking attack" showed a high level equivalent to the blockey as it dynamically generates keys based on the sensor data history (dynamically generated unpredictable sensor values). As shown in Sections 4.3.2 and 4.3.3, it is only possible in cases for malicious attacker make many brute force attacks to know the key generation information or to know the next key generated. It is impossible to find out the key generation information within the next key generated. Therefore, prevention of key generation information exposure and randomness/unpredictability of encryption key were also satisfied. The proposed method satisfied the availability and security of dynamic encryption keys in a low-power radio environment.

Table 6. Comparison table of related studies.

	OTP (S/Key) Kungpisdan et al. [29]	**Diffie–Hellman Diffie and Hellman [30]**	**Blockey WooSeung [23]**	**Proposed Method**
Availability	Encryption Key Generation Communication Method/Steps			
	Asynchronous /1 step	Synchronous /2 steps	Asynchronous /0 step	Asynchronous /1 step
	Encryption Key Recovery Communication Method/Steps			
	Synchronous /2 steps	Synchronous /2 steps	Synchronous /2 steps	Asynchronous /1 step
	Required Computational Cost/Time for Encryption Key Generation			
	Low	High	Low	Low
Security	Security Against Encryption Key Hijacking Attack			
	Vulnerable	Average	High	High
	Prevention of Key Generation Information Exposure			
	Prevention	Prevention	Prevention	Prevention
	Randomness/Unpredictability of Encryption Key			
	Randomness /Unpredictability	Randomness /Unpredictability	Randomness /Unpredictability	Randomness /Unpredictability

As shown in the Figure 16, our proposed method is within the scope of research on encryption key. For security in the real Internet of Things environment, access control and authentication on the Internet of Things should be considered [32,33].

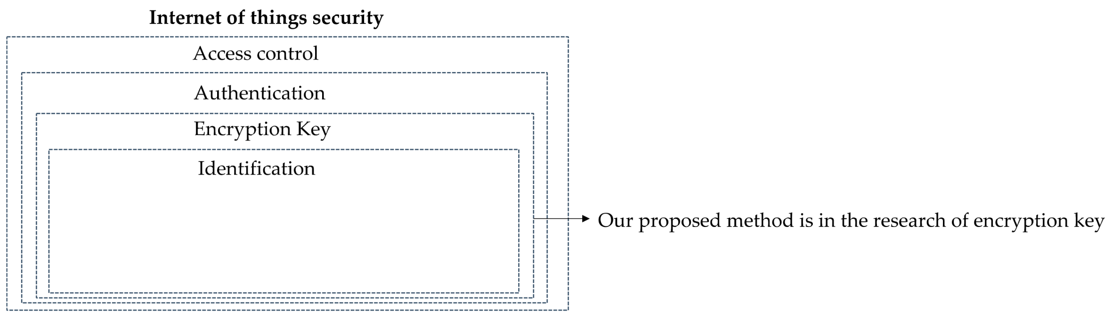

Internet of things security
Access control
Authentication
Encryption Key
Identification
Our proposed method is in the research of encryption key

Figure 16. Internet of Things security.

There are also problems to be solved in access control and authentication on the Internet of Things. The study by Buccafurri and Celeste [34] to improve the device authentication vulnerability of MQTT protocol, OTP-based authentication protocol using block chain was proposed. The study by Maissa et al. [35] decentralized light-weight access control model was proposed to address the issue of scalability of centralized access control. The study by Choudhary et al. [36] authentication and key management model considering the user's authentication of the device was proposed. Future research will refer to the above mentioned studies [34–36] to study how to extend the dynamic encryption key proposed in the paper to access control and authentication techniques. In addition, only theoretical verification of the randomness and unpredictability of the security aspects of the measures proposed in this study was carried out. Further studies would like to verify through the implementation of statistical randomness verification provided by NIST (National Institute of Standards and Technology) [37].

Author Contributions: Formal analysis, S.C.; Investigation, S.C. and D.K.; Methodology, S.C.; Software, S.C.; Validation, S.C.; Writing—original draft, S.C.; Writing—review & editing, D.K. and S.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Korea east-west power company.

Acknowledgments: This paper was supported in 2020 with the support of Korea east–west power company Research and Development (C-2019-6, Development and Implementation of IoT Security Enhancement Algorithm using Blockchain). This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2020-2017-0-01628) supervised by the IITP(Institute for Information & communications Technology Promotion).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Alghamdi, T.; Aiash, M.; Lasebae, A. Security Analysis of the Constrained Application Protocol in the Internet of Things. In Proceedings of the Second International Conference on Future Generation Communication Technology, London, UK, 12–14 November 2013.
2. Ghamari, M.; Arora, H.; Sherratt, R.; Harwin, W. Comparison of Low-Power Wireless Communication Technologies for Wearable Health-Monitoring Application. In Proceedings of the International Conference on Computer, Communications, and Control Technology (I4CT), Kuching, Malaysia, 21–23 April 2015.
3. Nikoukar, A.; Raza, S.; Poole, A.; Günes, M. Low-Power Wireless for the Internet of Things: Standards and Applications. *IEEE Access* **2018**, *6*, 67893–67926. [CrossRef]
4. Lin, J.; Li, F.; Luo, B. Cyber-Physical Systems Security—A Survey. *IEEE Internet Things J.* **2017**, *4*, 1802–1831.
5. M.Sadeeq, M.; M.Zeebaree, S.; Qashi, R.; Ahmed, S.; Jacksi, K. Internet of Things security A survey. In Proceedings of the International Conference on Advanced Science and Engineering (ICOASE), Duhok, Iraq, 9–11 October 2018.
6. Yangm, G.; Geng, G.; Du, J.; Liu, Z.; Han, H. Security threats and measures for the Internet of Things. *Qinghua Daxue Xuebao/J. Tsinghua Univ.* **2011**, *10*, 1335–1340.
7. El-hajj, M.; Fadlallah, A.; Maroun, A.; Serhrouchni, A. A Survey of Internet of Things (IoT) Authentication Schemes. *Sensors* **2019**, *19*, 1141. [CrossRef] [PubMed]
8. Ali, I.; Sabi, S.; Gullah, Z. Internet of Things Security, Device Authentication and Access Control: A Review. *Int. J. Comput. Sci. Inf. Secur. (IJCSIS)* **2016**, *14*, 456–466.
9. Oracevic, A.; Dilek, S.; Ozdemir, S. Security in Internet of Things: A Survey, International Symposium on Networks. In Proceedings of the Computers and Communications (ISNCC), Marrakech, Morocco, 16–18 May 2017.
10. Butun, I.; Österberg, P.; Song, H. Security of the Internet of Things: Vulnerabilities, Attacks, and Countermeasure. *IEEE Commun. Surv. Tutor.* **2020**, *22*, 616–644. [CrossRef]
11. Eltoweissy, M.; Moharrum, M.; Mukkamala, R. Dynamic Key Management in Sensor Networks. *IEEE Commun. Mag.* **2006**, *44*, 122–130. [CrossRef]
12. Sicari, S.; Rizzardi, A.; Miorandi, D.; Coen-Porisini, A. Internet of Things: Security in the Keys. In *The 12th ACM Symposium*; ACM: New York, NY, USA, 2016.
13. Zhang, Q.; Liang, Z.; Cai, Z. Developing a New Security Framework for Bluetooth Low Energy Devices. *Comput. Mater. Contin.* **2019**, *59*, 457–471. [CrossRef]
14. Dang, T.; Vo, H. Advanced AES Algorithm Using Dynamic Key in the Internet of Things System. In Proceedings of the International Conference on Computer and Communication Systems, Singapore, 23–25 February 2019.

15. Noura, H.; Chehab, A.; Couturier, R. Lightweight Dynamic Key-Dependent and Flexible Cipher Scheme for IoT Devices. In Proceedings of the Wireless Communications and Networking Conference (IEEE), Marrakesh, Morocco, 15–18 April 2019.
16. Adafruit Learning System. Introduction to Bluetooth Low Energy. Available online: https://cdn-learn.adafruit.com/downloads/pdf/introduction-to-bluetooth-low-energy.pdf (accessed on 20 December 2019).
17. Ngo, H.; Wu, X.; Dung Le, P.; Campbell, W. Dynamic Key Cryptography and Applications. *Int. J. Netw. Secur.* **2010**, *10*, 161–174.
18. Ahmed, M.; Sanjabi, B.; Aldiaz, D.; Rezaei, A.; Omotunde, H. Amirhossein Diffie hellman and its Use in Secure Internet Protocols. *Int. J. Eng. Sci. Innov. Technol.* **2012**, *1*, 69–73.
19. Begusic, D.; Rozic, N.; Stella, M. Speech recognition over Bluetooth ACL and SCO links: A comparison. In Proceedings of the Consumer Communications and Networking Conference (IEEE), Las Vegas, NV, USA, 6 January 2005.
20. Rondon, R.; Mahmood, A.; Grimaldi, S.; Gidlund, M. Understanding the Performance of Bluetooth Mesh: Reliability, Delay and Scalability Analysis. *IEEE Internet Things J.* **2020**, *7*, 2089–2101. [CrossRef]
21. Mazzenga, F.; Cassioli, D.; Loreti, P.; Vatalaro, F. Evaluation of packet loss probability in Bluetooth networks. In Proceedings of the IEEE International Conference on Communication Conference Proceedings, New York, NY, USA, 28 April–2 May 2002.
22. Yuan, C.; Chieh, H. A survey of key distribution in wireless sensor networks. *Secur. Commun. Netw.* **2011**, *7*, 2495–2508.
23. Soohwan, C.; SooYong, P.; Lee, R. Blockchain Consensus Rule Based Dynamic Blind Voting for Non-Dependency Transaction. *Int. J. Grid Distrib. Comput.* **2017**, *10*, 93–106.
24. WooSeung, L. Key Generation Algorithm based on Shared Message History for in-Vehicle Security Network. Available online: http://dcollection.sogang.ac.kr:8089/dcollection/public_resource/pdf/000000062077_20200929192442.pdf (accessed on 15 November 2019).
25. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. Available online: https://bitcoin.org/bitcoin.pdf321 (accessed on 15 May 2020).
26. Buterin, V. Emoreum: A Next-Generation Smart Contract and Decentralized Application Platform. Available online: https://emoreum.org/en/whitepaper/ (accessed on 20 May 2020).
27. Hyper Ledger Foundation, An Introduction to Hyperledger. Available online: https://www.hyperledger.org/wp-content/uploads/2018/08/HL_Whitepaper_IntroductiontoHyperledger.pdf (accessed on 15 January 2020).
28. Ahmed Samy, B.M.; Youssef, S.; El Gamal, A.; El Hadi, N. One-Time Password Authentication Techniques Survey. *Comput. Sci. Data Min. Knowl. Eng.* **2017**, *9*, 69–78.
29. Kungpisdan, S.; Le, P.; Srinivasan, B. Limited-Used Key Generation Scheme. In *International Workshop on Information Security Applications*; Springer: Berlin/Heidelberg, Germany, 2005.
30. Diffie, W.; Hellman, M. New directions in cryptography, Information theory. *IEEE Trans. Inf. Theory* **1976**, *22*, 644–654. [CrossRef]
31. Mishra, M.; Kar, J. A study on diffie-hellman key exchange protocols. *Int. J. Pure Appl. Math.* **2017**, *114*, 179–189. [CrossRef]
32. Misra, S.; Maheswaran, M.; Hashmi, S. System Model for the Internet of Things. In *Security Challenges and Approaches in Internet of Things*; Elsevier: Amsterdam, The Netherlands, 2016; pp. 5–17.
33. De, S.; Barnaghi, P.; Bauer, M.; Meissner, S. Service modeling for the Internet of Things. In Proceedings of the Federated Conference on Computer Science and Information, Szczecin, Poland, 18–21 September 2011.
34. Buccafurri, F.; Celeste, R. A Blockchain-Based OTP-Authentication Scheme for Constrainded IoT Devices Using MQTT. In Proceedings of the International Symposium on Computer Science and Intelligent Control (ICPS), Amsterdam, The Netherlands, 25–27 September 2019.
35. Maissa, D.; Sidi-Mohammed, S.; Mohamed Ayoub, M.; Mohamed Houcine, E. Decentralized Lightweight Group Key Management for Dynamic Access Control in IoT Environments. *IEEE Trans. Netw. Serv. Manag.* **2020**, *17*, 1742–1757.

36. Choudhary, K.; Gaba, G.S.; Butun, I.; Kumar, P. MAKE-IT—A Lightweight Mutual Authentication and Key Exchange Protocol for Industrial Internet of Things. *Sensors* **2020**, *20*, 5166. [CrossRef] [PubMed]
37. NIST. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Available online: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf (accessed on 20 June 2020).