symmetry

MDPI

*Article*

# LILP: A Lightweight Enciphering Algorithm to Encrypt Arbitrary-Length Messages

**Xing Zhang [1], Jian Chen [1], Tianning Li [2], Gang Dai [3] and Changda Wang [1,*]**

[1] School of Computer Science & Communication Engineering, Jiangsu University, Zhenjiang 212013, China
[3] Daquan Group Co., Ltd., Zhenjiang 212211, China
* Correspondence: changda@ujs.edu.cn

**Abstract:** The advancement of the Internet of Things (IoT) has promoted the development of embedded devices. It is important to ensure data transmission security on embedded devices with limited computing power and storage space. However, the traditional block encryption algorithm cannot run efficiently on embedded devices because of the large amount of computation. In this paper, a lightweight length-preserving-encryption algorithm (LILP) is proposed to convert an $n$-bit block cipher into a special block cipher that supports an arbitrary length of no less than $2n$ bits as input. LILP adopts the involution design method based on a Lai–Massey structure and lightweight components to adapt to the limited computing power of embedded devices. In particular, a lightweight compression function (LCF) is designed to process the data during iteration, which improves security without reducing the efficiency of the algorithm. The experimental results show that LILP is more efficient than traditional similar algorithms in encrypting data for resource-constrained devices while ensuring data security in the IoT.

**Keywords:** length-preserving encryption; lightweight cryptography; involution; internet of things



**Citation:** Zhang, X.; Chen, J.; Li, T.; Dai, G.; Wang, C. LILP: A Lightweight Enciphering Algorithm to Encrypt Arbitrary-Length Messages. *Symmetry* **2023**, *15*, 177. https://doi.org/10.3390/sym15010177

Academic Editor: Christos Volos

Received: 24 November 2022
Revised: 24 December 2022
Accepted: 4 January 2023
Published: 7 January 2023



## 1. Introduction

Internet of Things (IoT) is a network composed of many devices, such as physical gadgets, domestic appliances and microcontrol equipment. A large number of resource-constrained embedded devices are used in the IoT. The data generated by these devices cannot use traditional encryption algorithms that require large computing resources to ensure confidentiality. Therefore, lightweight encryption algorithms are required to ensure data security in resource-constrained devices. The existing blockciphers have limitations on block length. To design an encryption algorithm that accepts plaintext input of any length without selecting an encryption mode is more suitable for the IoT with diverse data, limited computing power and storage space.

The structure of IoT can be divided into the perception layer, the network layer and the application layer. As shown in Figure 1, wireless sensor networks (WSNs) play an important role in the perception layer. This has already been applied in many areas of the IoT, such as smart transportation, smart homes and smart healthcare. WSNs collect, process and transmit the information of the perceived area through a large number of sensor nodes.

For example, in a smart home, sensor nodes in a WSN collect the private data of the house and transmit them to the sink node. Sink nodes upload the sensing information to an IoT cloud server. Users can download and view private family data by using mobile phones when they leave home. Data containing personal privacy may be transmitted through an untrusted network layer or stored in untrusted cloud services. The confidentiality of sensitive data is potentially subject to cyber attacks [1,2]. In addition, the perception layer also has the risk of being attacked [3]. Hence, the perception layer requires security measures for information acquisition in the IoT [4].

However, the cost and performance of protecting data in the perception layer are critical issues that must be considered in combination. Due to the limited computing

capabilities and power of components in WSNs, ensuring the communication security of WSNs requires high performance costs [5]. For example, the main frequency of the remote terminal unit (RTU) used as the sink node in smart transportation is only 48 Mhz, and the memory is only 1 M byte.

Standard encryption algorithms used in RTU can provide high security, such as symmetric encryption algorithm (e.g., AES) and asymmetric encryption algorithm (e.g., Elliptic Curve Cryptography). Since these algorithms involve a great deal of computation, the limited processing and storage capacity of sensors cannot effectively support them. This fact drives the demand for more secure, less computational and more flexible encryption algorithms.

A variety of lightweight symmetric block ciphers have been designed. Lightweight symmetric cryptography in resource-constrained devices is a hotspot in current research of information safety [6]. However, existing lightweight symmetric cryptography research mainly focuses on the optimization of single-block encryption. The processing methods between blocks with a large impact on efficiency and security have not been emphasized. The traditional processing methods for encrypting a long message are composed of encryption modes and length-preserving-encryption algorithms.

The classic encryption modes include Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB) and Counter mode (CTR) in the DES modes of operations [7]. Generally, the bit length of plaintext is not an integer multiple of the block length. The ciphertext is longer than the plaintext if a mode similar to ECB is used. It means that more storage and transmit space is needed to temporarily store these data on embedded devices. In addition, it cannot meet the existing security requirements if the CTR mode is used.

The classic length-preserving-encryption algorithms include Arbitrary Block Length Mode (ABL) [8], a Variable-Input-Length Enciphering Mode (HCTR) [9] and the Extended Codebook Mode (XCB) [10], which allow devices to store the ciphertext without additional storage space. For example, XCB can be used to encrypt an arbitrary block length and provide the best possible security in the light of disk-block encryption. However, such methods are devised for common devices but not resource-constrained devices.

Therefore, the problems come from two aspects. One is how to ensure that as much data as possible can be stored on the resource-constrained devices, the other is how to make the encryption algorithm for processing large amounts of data run efficiently in such devices. Focusing on these questions, we design a scheme that uses length-preserving-encryption algorithms to provide a variety of optional input lengths and reduce pressure on data transmission in the IoT. In addition, by converting the existing length-preserving-encryption algorithm, a lightweight version can be obtained, which reduces the consumption of computing resources.

We propose a lightweight length-preserving symmetric enciphering algorithm (LILP), which is based on the Lai–Massey [11] structure. LILP accepts any plaintext whose length is greater than or equal to the length of two blocks. LILP consists of a compression function, a counter mode (CTR), whitening key additions and S-boxes. Specifically, the main contributions of LILP in this paper are as follows.

- The decryption process of LILP can reuse its encryption process because it uses symmetric structures and involuntary components.
- A lightweight compression function (LCF) is carefully designed for running on 32-bit platforms, which gives LILP an advantage over existing length-preserving-encryption algorithms in terms of speed and cost.
- LILP can accept different bit-level input sizes so that the caller has flexibility in choosing the size of the encrypted data.

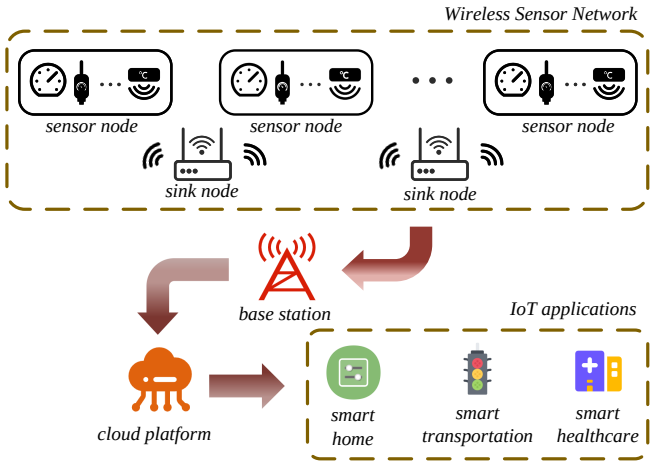


**Figure 1.** Application of a wireless sensor network in IoT.

The remainder of this paper is organized as follows. In Section 2, we introduce the LILP. In Section 3, the security of LILP is discussed. In Section 4, we present the software implementation results of LILP. Finally, we conclude our work in Section 5.

## 2. The Proposed Encryption Algorithm

LILP is a lightweight encryption algorithm that generates multiple subkeys from one master key for use by various components. The length of the master key is designed as double the key size of an underlying block encryption algorithm. LILP uses a symmetrical structure, which can be divided into five layers in total—namely, two XOR layers, two Lai–Massey layers and a CTR layer. Figure 2 shows the encryption of LILP.

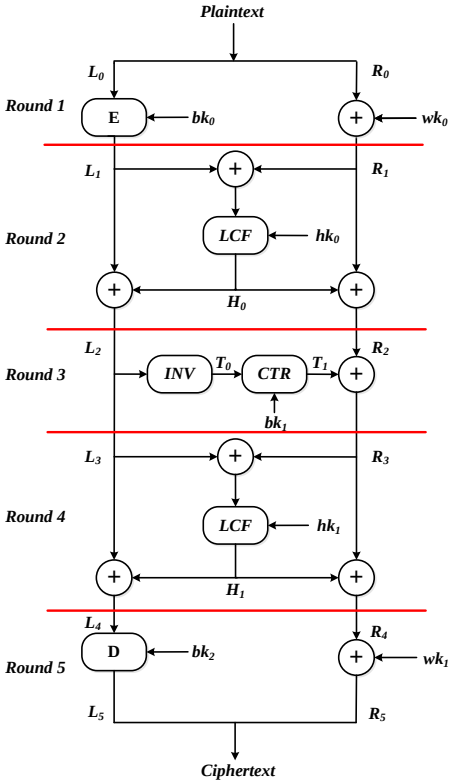


**Figure 2.** The encryption process of LILP.

### 2.1. Encryption Algorithm

The description of the notation used in the proposed LILP is listed in Table 1. LILP makes use of a pair of block encryption and decryption algorithms and a specific compression function $H$. Suppose that the blockcipher is $E : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$, then $LILP[E,H]$ is

$$LILP[E,H] : \{0,1\}^{2k} \times \{0,1\}^{\geq 2n} \rightarrow \{0,1\}^{\geq 2n}, \quad (1)$$

where $\{0,1\}^{\geq 2n} = \bigcup_{m \geq 2n} \{0,1\}^m$. Algorithm 1 shows $LILP[E,H]$ in detail.

**Table 1.** Description of the notations used in the proposed LILP.

| Symbol | Description |
|---|---|
| $\|$ | concatenation of two strings. |
| $\ll n$ | left shift by $n$ bits. |
| $\gg n$ | right shift by $n$ bits. |
| $\oplus$ | bitwise exclusive-OR operation. |
| $a_{(b)}$ | $b$ denotes the bit length of $a$. |
| $0 \cdots 0$ | a string of 0s. |
| $\{0,1\}^k$ | the set of all strings of the length of $k$. |

**Algorithm 1** The proposed LILP encryption

**Input:** $P, K$
**Output:** $C$
1: $(bk_0, bk_1, bk_2, fk_0, fk_1) \leftarrow KeyForBC(K)$
2: $(hk_0, hk_1) \leftarrow KeyForHash(K)$
3: $(wk_0, wk_1) \leftarrow KeyForWhiting(fk_0, fk_1, i)$
4: $L_0 \leftarrow P[0, n-1]$
5: $R_0 \leftarrow P[n, len(P)-1]$
6: $(L_1 \parallel R_1) \leftarrow XORLayer(E, L_0, R_0, bk_0, wk_0)$
7: $(L_2 \parallel R_2) \leftarrow LaiMasseyLayer(L_1, R_1, hk_0)$
8: $(L_3 \parallel R_3) \leftarrow CTRLayer(L_2, R_2, bk_1)$
9: $(L_4 \parallel R_4) \leftarrow LaiMasseyLayer(L_3, R_3, hk_1)$
10: $(L_5 \parallel R_5) \leftarrow XORLayer(D, L_4, R_4, bk_2, wk_1)$
11: **return:** $(L_5 \parallel R_5)$

We provide a concrete instance that takes the Piccolo [12] block cipher as the underling block encryption algorithm. A variable-input-length block cipher that accepts message sizes larger than or equal to 128 bits is created. Piccolo is a variant of a generalized Feistel network. It supports 64-bit block size with 80-bit and 128-bit keys through 25 and 31 rounds, respectively.

The 80-bit key version of Piccolo was chosen in our design. Thus, a 160-bit master key, twice key size of Piccolo, is used to generate a series of subkeys. The length of plaintext is at least 128 bits. A plaintext $P$ is divided into two parts $L_0$ and $R_0$, in which $L_0$ is the first 64 bits of the plaintext while $R_0$ is the remainder of the plaintext. Then, five layers of operations continue to perform.

In the first layer, $L_1$ is obtained from running a block encryption on $L_0$. $R_1$ is obtained by XOR $R_0$ with whitening key $wk_0$ at the same time. The whitening key $wk_0$ is obtained

by iteratively using Piccolo with reduced rounds. The detailed process is expressed by the formula as follows:

$$\begin{cases} P = L_0 \parallel R_0, \\ L_1 \parallel R_1 = (E(bk_0, L_0)) \parallel (R_0 \oplus wk_0). \end{cases} \tag{2}$$

The Lai–Massey structure is applied in the second layer, which is characterized by involution. $L_1$ is first filled by padding zero bits. The length of $L_1$ after filling is equal to the length of $R_1$. Then, a result of XOR is input into $LCF$. Assuming that the value generated by compression function is $H_0$, then $H_0$ and $H_0^{'}$ is XOR with $L_1$ and $R_1$, respectively. An operation similar to filling $L_1$ needs to be performed on $H_0$ before the XOR operation with $R_1$. The process of the second layer is represented by the formula as follows:

$$\begin{cases} L_1^{'} = L_1 \parallel 0 \cdots 0, \\ H_0 = LCF(hk_0, L_1^{'} \oplus R_1), \\ H_0^{'} = H_0 \parallel 0 \cdots 0, \\ L_2 \parallel R_2 = (L_1 \oplus H_0) \parallel (R_1 \oplus H_0^{'}). \end{cases} \tag{3}$$

In the third layer, a initial vector $T_0$ is formed by inputting $L_2$ into the nonlinear component $INV$. It refers to sixteen identical and involutive S-boxes. Enough intermediate state $T_1$ is formed after $T_0$ is input into the $CTR$ layer. A partial input $R_3$ of the next round is obtained from $T_1$ and $R_2$. The detailed process of the third layer is shown by the formula as follows:

$$\begin{cases} T_0 = INV(L_2), \\ T_1 = CTR(bk_1, T_0), \\ L_3 \parallel R_3 = L_2 \parallel (R_2 \oplus T_1). \end{cases} \tag{4}$$

The operation in the fourth layer is similar to the second layer. The XOR of left and right parts is input into the compression function to obtain a 64-bit compression value. The value is mixed into the left and right parts. The process of the fourth layer is expressed by the formula as follows:

$$\begin{cases} L_3^{'} = L_3 \parallel 0 \cdots 0, \\ H_1 = LCF(hk_1, L_3^{'} \oplus R_3), \\ H^{'} = H_1 \parallel 0 \cdots 0, \\ L_4 \parallel R_4 = (L_3 \oplus H_1) \parallel (R_3 \oplus H_1^{'}). \end{cases} \tag{5}$$

The block decryption algorithm $D$ and whitening key addition are used in the fifth layer. The output of the block decryption algorithm and the results of XORing $R_4$ with whitening key $wk_1$ are spliced together to obtain the final ciphertext $C$. The process of the last layer is formulated as follows:

$$\begin{cases} L_5 \parallel R_5 = (D(bk_3, L_4)) \parallel (R_4 \oplus wk_1), \\ C = L_5 \parallel R_5. \end{cases} \tag{6}$$

### *2.2. Key Schedule*

As shown in Figure 3, the key schedule divides a 160-bit primary key $PK$ into four 40-bit subkeys $k_{i(40)} (0 \leq i < 4)$, which provides three $bk_{i(80)} (0 \leq i < 3)$ and two $fk_{i(80)} (0 \leq i < 2)$ as follows:

$$\begin{cases} bk_0 = k_0 \parallel k_1, \quad bk_1 = k_0 \parallel k_2, \quad bk_2 = k_0 \parallel k_3, \\ fk_0 = k_1 \parallel k_2, \quad fk_1 = k_1 \parallel k_3. \end{cases} \tag{7}$$

In order to generate the keys required for the compression function, $PK$ is denoted as five 32-bit subkeys $e_{i(32)}(0 \leq i < 5)$ as $PK = e_0 \parallel e_1 \parallel e_2 \parallel e_3 \parallel e_4$. Then, the generation method of $hk_{0(32)}$ and $hk_{1(32)}$ is expressed by the formula as follows:

$$\begin{cases} hk_0 = e_0 \oplus e_1 \oplus e_2 \oplus e_3 \oplus e_4, \\ hk_1 = (hk_0 \ll 1) \oplus (hk_0 \gg 1). \end{cases} \tag{8}$$

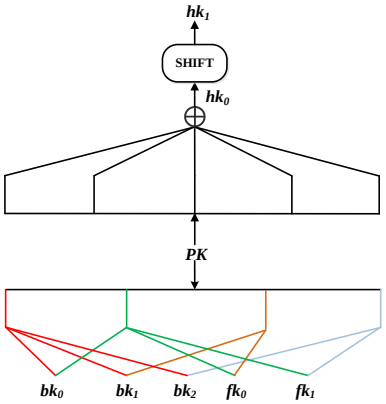


**Figure 3.** The key schedule of LILP.

A variable-length whitening key $wk_i(0 \leq i < 2)$ is generated by using a reduced-round block cipher $RE$. The so-called reduced-round block cipher refers to a block cipher that reduces the number of encryption rounds. $R_0$ can be divided into $m$ groups according to 64 bits as a group. The last group, if less than 64 bits, are also counted as a group. A constant block $0^{64}$ is encrypted by using the reduced-round block cipher $RE$. $0^{64}$ is a 64-bit string of 0s.

The results of the encryption are used as the first part of whitening key. The remaining parts of the whitening key are obtained from performing iterative encryption. The parameter $i$ of $RE$ refers to number of generation reduction rounds. The value is recommended as 3 during our experiments. We use $Pad(X)$ to append 0s to turn the bit length of $X$ into 64. The specific process of the generating whitening key $wk_0$ is as follows:

$$\begin{cases} vt_0 = RE(0^{64}, fk_0, i), \\ vt_j = RE(vt_{j-1}, fk_0, i) \quad (1 \leq j \leq m-1), \\ wk_0 = vt_0 \parallel vt_1 \parallel \cdots \parallel Pad(vt_{m-1}). \end{cases} \tag{9}$$

The process of generating $wk_1$ is essentially the same as $wk_0$. The difference is that $vt_{m-1}$ in $wk_0$ is used instead of $0^{64}$ in calculation. A different subkey $fk_1$ is also used.

$$\begin{cases} vp_0 = RE(vt_{m-1}, fk_1, i), \\ vp_j = RE(vp_{j-1}, fk_0, i) \quad (1 \leq j \leq m-1), \\ wk_1 = vp_0 \parallel vp_1 \parallel \cdots \parallel Pad(vp_{m-1}). \end{cases} \tag{10}$$

### *2.3. LCF*

The execution flow of LCF is presented in Figure 4. The design inspiration of LCF comes from the Murmurhash2B function. Diffusivity is the main indicator to be considered when modifying the Murmurhash2B function to achieve the LCF. The implementation cost of LCF is also concerned. Currently, many embedded chips have multipliers. However, there are still many MCUs that do not have multipliers due to cost constraints. Multiplication needs to be implemented through addition and shift instead. In short, the advantage of using addition outweighs multiplication in terms of the cost and performance.

Therefore, we suggest replacing certain multiplication operators in the Murmurhash2B function with the modular addition of a lightweight component. A search algorithm is devised to realize the above idea. This algorithm attempts to use as many modular addition operations as possible in LCF, rather than multiplication operations. The search algorithm is described as Algorithm 2.

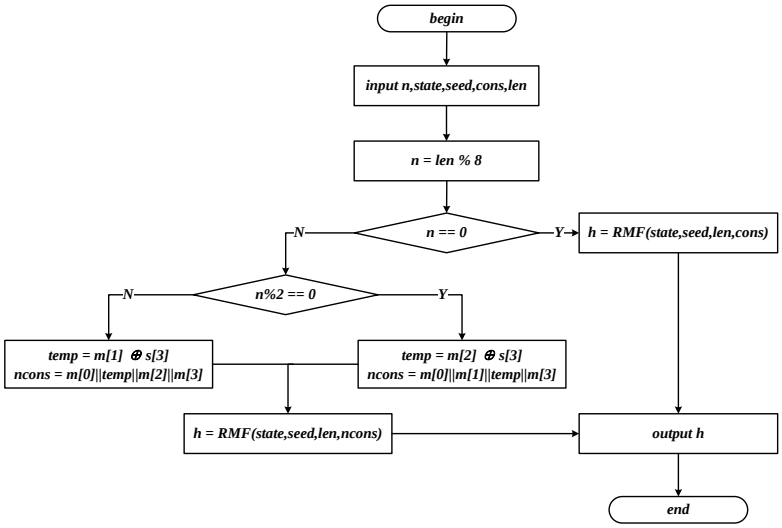


**Figure 4.** The execution flow of LCF.

**Algorithm 2** The search algorithm

```
for multis = 0 to 7 do
    success = 0
    for cond = 0 to 7 do
        start = 64 + cond
        end = 1920 + cond
        for const = 0x5b000000 to 0x5c000000 do
            flag = 0
            min = 1
            for i = start to end do
                if i%8 == 0 then
                    rate = CheckDuffsion(i)
                    if rate < 0.45 then
                        break
                    else
                        flag++
                        min = MinRate(min, rate)
            Record(cond, constant, min)
            count++
        Recordn(cond, count)
    CheckSuccess(multis)
```

The goal of the search algorithm is to make the function LCF obtain a better diffusion rate when processing data of different lengths. The results of the search algorithm are constants that can be used in LCF. Constants within a certain range are searched in Algorithm 2 to make the diffusion rate of LCF reach the target for each data length. In other words, LCF can choose different constants according to the length of the data.

In our experiment, suppose that the multiplication operations can be reduced by 4 to 14 times. *multis* represents the number of multiplication used in LCF. The data lengths processed by LCF range from 64 bits to 1927 bits, and these will be divided into eight groups.

Each group is characterized by the same results when the elements in the group are treated with mod 8. For each group, constants from 0x5b000000 to 0x5c000000 are sought when the number of multiplication operations used is determined. The function *Record* is used to record the group number, constant and lower limit of the diffusivity if the constant makes all points in a group meet the diffusivity condition. *Recordn* is used to record the number of constants in each group that meet the conditions. *CheckSuccess* is used to check the constant results for each group after searching the eight groups.

It is possible to use modular addition instead of multiplication if all eight groups have results. A constant is selected from the numerous constants after the search algorithm is completed. Not only can this make the LCF obtain an equilibrium diffusion rate but it can also deduce other constants from it. In other words, the selected constants can make the diffusion effect of LCF independent of the data length.

Finally, the fourteen multiplication operations in Murmurhash2B are simplified into four multiplication operations through experiments. The corresponding constants are analyzed to obtain the constant 0x5be5e995 used to design LCF. The selected constant can make the lower limit of diffusivity for LCF reach 0.47. In particular, the diffusivity of LCF can reach 0.5 when LCF uses eight times the data. In conclusion, the proposed LCF processes data faster and more cheaply than Murmurhash2B. The performance comparison between LCF and other functions is in Section 4. LCF is described in Algorithm 3.

LCF receives two parameters: key (*seed*) and data (*state*). A seed is provided by the key processing function, which is divided into four words. The bit length of each word is 8 bits. The bit length of the state and seed together determine the constant to be used when calling function RMF in LCF. *cons* can evolve 32 values under different conditions. This is used directly as an argument to RMF when the bit length of state is an integer multiple of 8. A conditional judgment is executed in other cases. The first or second word in *cons* is changed to create a new constant (*ncons*) after the condition is determined. *ncons* is generated by XOR operation on the third byte of *seed* and *cons*. LCF has a 64-bit compressed value by calling RMF after determining the constant. RMF is described in Algorithm 4.

Each basic operation processes 32-bit data in RCF. $h_0$ and $h_1$ are generated by processing every eight bytes of the state through modulo addition, shift and XOR operations. The first four bytes of a group with less than eight bytes are processed similarly to the previous step. The last three bytes are connected and added to $h_1$. Then, $h_0$ and $h_1$ are mixed by XOR, shift and multiplication operations. The final value *res* is the combination of $h_0$ and $h_1$.

**Algorithm 3** The lightweight compress function LCF

**Input:** $state, seed$
**Output:** $h$
$seed = s[0] \parallel s[1] \parallel s[2] \parallel s[3]$
$cons = m[0] \parallel m[1] \parallel m[2] \parallel m[3] = 0x5be5e995$
$len = Length(state)$
$n = len\%8$
**if** $n \neq 0$ **then**
    **if** $n\%2 \neq 0$ **then**
        $temp = m[1] \oplus s[3]$
        $ncons = m[0] \parallel temp \parallel m[2] \parallel m[3]$
    **else**
        $temp = m[2] \oplus s[3]$
        $ncons = m[0] \parallel m[1] \parallel temp \parallel m[3]$
    $h = RMF(state, seed, len, ncons)$
**else**
    $h = RMF(state, seed, len, cons)$
**return:** $h$

**Algorithm 4** The revised murmurhash2B function RMF

**Input:** $state, seed, len, m$
**Output:** $res$
$h_0 = seed \oplus len$
$h_1 = 0$
$r = 24$
**while** $len \geq 8$ **do**
  $k_0 = NextWord(state)$
  $k_0 = k_0 + m, \quad k_0 = k_0 \oplus (k_0 >> r) + m$
  $h_0 = (h_0 + m) \oplus k_0$
  $len = len - 4$
  $k_1 = NextWord(state)$
  $k_1 = k_1 + m, \quad k_1 = k_1 \oplus (k_1 >> r) + m$
  $h_1 = (h_1 + m) \oplus k_0$
  $len = len - 4$
**if** $len \geq 4$ **then**
  $k_2 = NextWord(state)$
  $k_0 = k_0 + m, \quad k_0 = k_0 \oplus (k_0 >> r) + m$
  $h_0 = (h_0 + m) \oplus k_0$
  $len = len - 4$
$temp = state[2] \parallel state[1] \parallel state[0]$
$h_1 = (h_1 \oplus temp) + m$
$h_0 = h_0 \oplus (h_1 \gg 18), \quad h_0 = h_0 * m$
$h_1 = h_1 \oplus (h_1 \gg 22), \quad h_1 = h_1 * m$
$h_0 = h_0 \oplus (h_1 \gg 17), \quad h_0 = h_0 * m$
$h_1 = h_1 \oplus (h_1 \gg 19), \quad h_1 = h_1 * m$
$res = h_0 \| h_1$
**return:** $res$

### *2.4. Decryption Algorithm*

The shared key between the encrypting party and the decrypting party can be exchanged through a secure key exchange protocol. There are many such key exchange protocols, such as those proposed in [13–15]. The encryptor sends the $PK$ to the receiver of the ciphertext through a secure key exchange protocol. The receiver of the ciphertext uses the $PK$ to generate a series of subkeys for decryption. The decryption operation of the proposed LILP is the same as the encryption operation. However, each subkey needs to be applied in reverse order when decrypting. Assuming that the order of keys used in encryption is $bk_0, wk_0, hk_0, bk_1, hk_1, bk_2, wk_1$, and then the order of keys used in decryption is $wk_1, bk_2, hk_1, bk_1, hk_0, wk_0, bk_0$.

## 3. Security Analysis

### *3.1. Avalanche Effect*

This part analyzes the security of LILP by diffusion and confusion tests. In the first step, a plaintext is created randomly and changes a bit randomly to create another plaintext. Then, the two plaintexts are encrypted by LILP to generate two results for comparison to obtain the diffusion degree. Finally, the above steps are repeated 100,000 times to calculate the average diffusivity. For plaintexts with lengths from 128 to 1499 bits, the average diffusion rates under different plaintext lengths are calculated, respectively. The test results are shown in Figure 5.

The degree of confusion is another important test to measure the security of an algorithm. Similar to the steps of the diffusion test, a key is generated at random. A bit of the key is randomly changed to generate another key. Encryption is repeated several times to calculate the average confusion rate. The test result is shown in Figure 6. According to the

results shown in Figures 5 and 6, the diffusion and confusion rates of LILP can both reach 0.5. In other words, LILP is safe in terms of randomness defined by the avalanche effect.

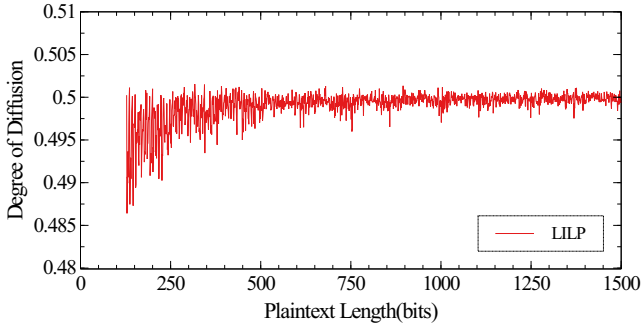


**Figure 5.** Diffusion analysis of the proposed LILP.

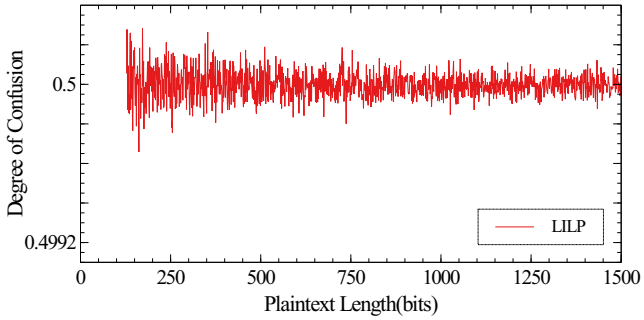


**Figure 6.** Confusion analysis of the proposed LILP.

In addition, different key schedules are studied according to the degree of confusion, which can be divided into four cases.

- The first case is that the underlying block encryption and decryption algorithm uses the same key as the CTR mode. A fixed key is used for a compression function. In this case, the confusion rate of an algorithm can only reach 0.33.
- The second case is that the underlying block encryption and decryption algorithm mode use different keys. The compression function uses a fixed key. However, the confusion rate cannot be improved in this case.
- The third case is that a master key is simply processed to generate keys used by the compression function. In this case, the confusion rate will be significantly improved.
- The last case is that the key schedule function is precisely devised so that a change of the master key may result in as many changes of subkeys as possible. Then, the confusion rate of an algorithm in this case can reach about 0.5.

### *3.2. Discussion*

In terms of security, the LILP has a structural design similar to XCB. Therefore, the security analysis of XCB mode can be directly applied to LILP mode. In [16], XCB is proven to be secure as long as a block cipher is used in the security model based on a specific reduction that can be considered as a secure pseudorandom permutation. In this instance, we can assume that LILP is also safe. A different design in LILP is to add two whitening keys.

Key whitening includes the steps of combining the key with the state part before the first round of encryption and after the last round of encryption in LILP. This is intended to increase the complexity of violent attacks and increase the effective size of the key without requiring significant changes to the algorithm. In the structure of the XCB, the right part of the state is placed in the compression operation, which directly influences the compression value. LILP places the entire state into a compression function to distinguish it from XCB.

This practice makes the hash value more unpredictable. In other words, any change in an original state results in a different compression value.

## 4. Performance Analysis

### *4.1. Overall Comparison*

As shown in Table 2, length-preserving-encryption algorithms can accept input of any length. Some components of the length-preserving algorithms can be operated in parallel. In addition, algorithms using the Luby–Rackoff structure do not input the entire plaintext into the compression function, which affects the diffusion effect of the algorithms. The Lai–Massey structure adopted by LILP can solve this problem. It is worth noting that the GHash-like compression functions used by ABL, XCB and HCTR need to implement multiplication over finite fields. A simple implement of multiplication is much slower than one AES call [17]. However, the compression function LCF used by LILP does not require such a large computational cost. In order to adapt to resource-constrained devices, LCF also takes cost, security and performance into consideration comprehensively.

**Table 2.** The design criteria of ABL, HCTR, XCB, HCH and LILP.

| | Compression Method | Variable Input Length | Parallelizable | Structure |
|---|---|---|---|---|
| ABL [8] | GHash | √ | Partially | Luby–Rackoff |
| HCTR [9] | GHash | √ | Partially | Luby–Rackoff |
| XCB [10] | AXUHash | √ | Partially | Luby–Rackoff |
| HCH [17] | Wegman-Carter | √ | Partially | Luby–Rackoff |
| LILP | LCF | √ | Partially | Lai–Massey |

We implemented LILP on the 1.6 GHz Intel(R) Core i5-8250U CPU, which uses the Intel C++ compiler to run a 64 bit operating system. In addition, there are devices used for data acquisition, control and communication functions in the perception layer. They often have more abundant computing resources and storage resources when compared with sensor nodes, such as RTU [18]. The frequency range of the embedded processing chips typically used is from tens of MHZ to hundreds of MHZ [19–21]. Compared with the dominant frequency of GHZ on the server, the data processing capability is still relatively weak. Therefore, we also implemented LILP on RTU, which uses a 32-bit 48 MHz ARM processor with 32 single byte general-purpose registers, 1 M byte SRAM and 8 M byte programmable flash memory.

First of all, LILP is a lightweight and improved version of XCB. We compare the encryption and decryption speed of LILP with XCB. We prepare 10 GB and 1 GB of data for each plaintext length on PC and RTU, respectively, to calculate the average running time of LILP and XCB. The comparison results are shown in Figure 7, and these indicate that LILP is faster than XCB in processing data on both PC and RTU.

In addition, we select several existing length-preserving-encryption algorithms of the same type as LILP to compare the encryption and decryption speed. Since algorithms, such as ABL and HCTR, do not have a specific key schedule function, for the sake of fairness, all keys required by the algorithm are prepared in advance. The running time of the algorithm on RTU is also calculated and compared. As shown in Figure 8, we observed that LILP shows better performance to achieve length-preserving encryption of 128-bit to 1499-bit data. The comparison with other ciphers for encrypting 1000-bit plaintext is shown in Table 3. In terms of the throughput, LILP is faster than ABL, HCTR and XCB.

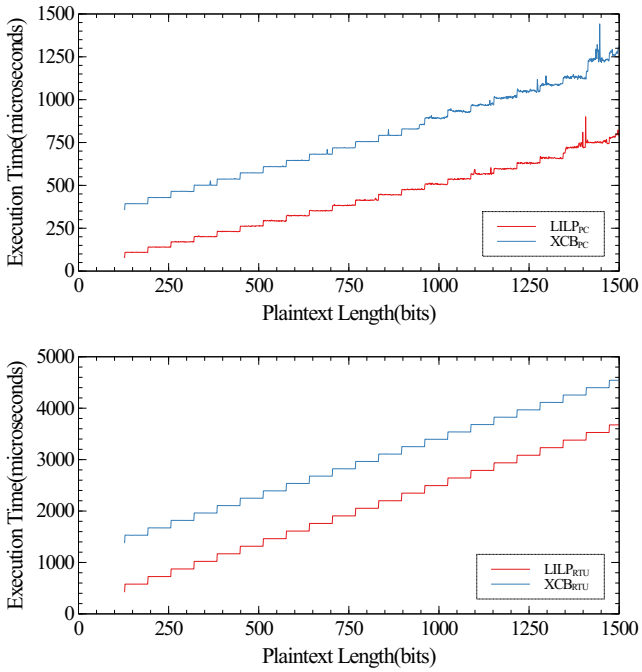


**Figure 7.** Comparisons of the execution time of LILP and XCB.

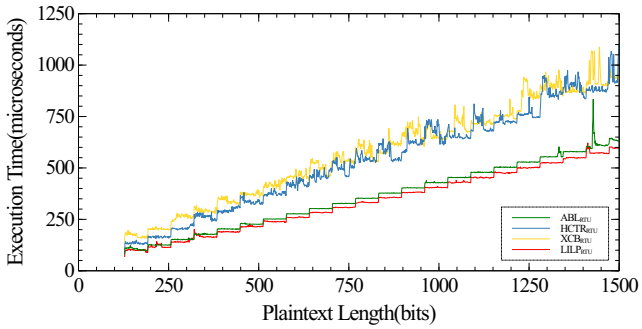


**Figure 8.** Comparison of the execution time of ABL, HCTR, XCB and LILP.

**Table 3.** Comparison of CMC ABL, HCTR, XCB and LILP.

| Algorithms | Execution Time (Microseconds) | Throughout (kbit/s) |
|---|---|---|
| ABL | 428 | 2281 |
| HCTR | 712 | 1371 |
| XCB | 643 | 1518 |
| LILP | 405 | 2411 |

### *4.2. Comparison of the Key Schedule*

We suppose that the underlying block encryption algorithm can generate 64-bit ciphertext by using 80-bit keys after 25 rounds of transformation. The key schedule algorithm of XCB uses an 80-bit master key to generate a 64-bit and three 80-bit subkeys. The 64-bit subkey is ciphertext generated by encrypting all zero constants with a block cipher. The 80-bit subkeys are results of encrypting different constants by calling the block encryption algorithm twice. The advantage of XCB using such a key schedule is to save space for storing keys.

However, if such a key schedule is used in resource-constrained embedded devices, it requires seven times the block encryption algorithms to generate four subkeys, which will

undoubtedly reduce the operating efficiency of the entire system. Therefore, it is necessary to run a simple and fast key schedule function on resource-constrained embedded devices to ensure a certain degree of security.

In this paper, the reduced round block encryption algorithm is used to generate the whitening keys. The keys required for the block encryption algorithm and compression function are realized through the operations of grouping, splicing and shifting. The time required to generate a whitened key varies with the length of plaintext. The advantage of using the reduction round method is to reuse existing components. In addition, whitening keys can be generated safely and quickly when plaintext of a certain length range is encrypted.

For example, compared with the large amount of time spent on the XCB key schedule, the time for LILP to complete the key schedule in the case of plaintext lengths from 448 to 640 bits is equivalent to the time of calling the block cipher twice. It can be seen from Figure 9 that, when the plaintext length is less than 1600 bits, the execution time of LILP is significantly shorter than XCB in the key schedule. Some security risks may be hidden in a length-preserving-encryption algorithm without a key schedule function. For example, the ciphertext generated by HCH [17] encryption does not have strong randomness without processing a master key.

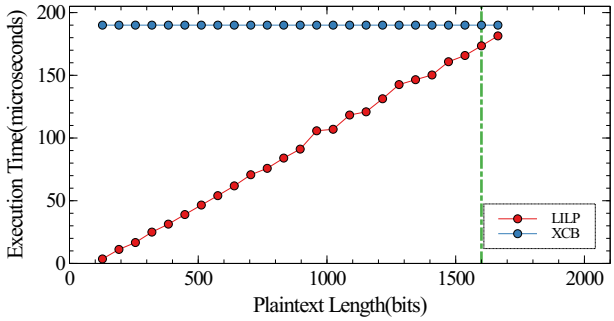


**Figure 9.** Comparison of the execution time of key schedules.

### *4.3. Comparison of Compression Function*

Since the choice of compression function has a great impact on the performance and security of the proposed LILP, it is necessary to analyze and compare compression functions. The specific results of analysis and comparison are as follows. First, the performance of three compression algorithms—namely, AXUHash, GHash and Murmurhash2B are compared. A special AXU (AlmostXorUniversal) function can accept $X$ of any length by dividing $X$ into $X = X_1 \cdots X_m$. If the length of $X_m$ is less than $n$, then enough 0 bits are filled in the end of $X_m$ to make the bit length of $X_m$ equal to $n$. $|X|$ refers to the bit length of $X$. $h$ is a parameter similar to the key that needs to be specified separately. A special AXUHash is defined as follows:

$$H_h(X) = X_1 \cdot h^{m+1} \oplus \cdots \oplus X_m 0^* \cdot h_2 \oplus |X| \cdot h \tag{11}$$

This compression function is based on multiplication and addition over finite fields. A fixed-length value is formed by dividing input into blocks for absorption. The advantage of the above compression functions is that it can be implemented efficiently on a high-performance server. However, its disadvantage is also obvious. If the platform uses 32-bit or less algorithms, there is no data type long enough to effectively implement operations over finite fields. Therefore, the implementation of multiplication is more complicated, which will greatly reduce the operational efficiency of this compression function. Similar to AXUHash, GHash is also based on polynomial operations over finite fields. The specific calculation details of GHash are shown in [8].

In addition, the influence of different compression functions on the diffusion effect of the algorithm is also studied. When all inputs are 0, AXUHash and GHash will not work,

which means that the compressed value will also be 0 or a simple value. This is because any element in a finite field multiplied by 0 will be 0.

The experimental results show that Murmurhash2B has a good diffusion effect. It uses XOR, shift, normal addition and multiplication to obtain a 64-bit value [22]. The running time of these three compression functions is tested multiple times with 5 GB data. The average running time obtained is shown in Table 4. In addition, the advantage of using Murmurhash2B is that it can directly process the byte stream submitted by the bottom layer. The byte stream needs to be formatted before using GHash and AXUHash, which reduces the running efficiency of the whole algorithm to a certain extent.

**Table 4.** The execution time of the compression function.

| Functions | Execution Time (Microseconds) |
|---|---|
| GHash | 151,947,619 |
| AXUHash | 138,007,244 |
| Murmurhash2B | 19,061,279 |
| LCF | 17,017,134 |

### *4.4. Application*

The proposed LILP is applied to protect sensor data in WSNs. In addition, a lightweight key exchange security framework [23] is used for key transmission. WSNs consist of sensor nodes, such as temperature sensors, soil sensors, humidity sensors, light sensors and soil pH sensors. These sensor nodes are installed manually on the WSN to collect environmental data before transmission to the sink node. Then, the sink node transmits the data to the user through a common channel.

Users make choices and take necessary actions on sensor data property based on the environmental facts obtained. RTU is used to collect sensor data in a monitoring area with 100 monitoring points. Each monitoring point uses temperature and humidity sensors, light sensors and carbon dioxide sensors to monitor the environmental information of the crops. The sensor data is encrypted on the RTU and transmitted to the cloud server through the 4G network.

We stipulate that the temperature and humidity information occupies 4 bytes, the light information occupies 3 bytes, the carbon dioxide information occupies 2 bytes, and the location code occupies 1 byte. When RTU reads sensor data from 100 nodes, the data size to be encrypted is 1000 bytes. The experimental results show that it takes 0.02 s to call LILP to encrypt such sensor data. The advantage of LILP in encryption speed makes it have less impact on other services running in the RTU. In addition, LILP has the advantage of adapting to sensor data of different lengths. For example, sometimes adding monitoring nodes or damaging sensors will cause changes in the amount of data.

## 5. Conclusions

In this paper, we first analyzed certain traditional algorithms that are not suitable for encrypting large amounts of data on resource-constrained devices. Then, a lightweight encryption algorithm LILP was proposed. LILP uses a provably secure Lai–Massey structure and two variable length whitening keys to ensure its security. The most significant characteristic of LILP is that the encryption and decryption are consistent, which enables LILP to reuse code.

In addition, a lightweight compression function LCF was designed to give LILP fast encryption and decryption speeds with a low cost. Finally, we tested the performance of LILP in terms of the execution time and throughput. The execution time for LILP to encrypt an 1000-bit plaintext was 405 microseconds with the throughput of 2411 kbps. The results show that LILP was superior to traditional algorithms in the speed of encryption and decryption for lightweight enciphering.

**Author Contributions:** Conceptualization, X.Z. and J.C.; Methodology, X.Z.; Software, J.C.; Validation, T.L., G.D. and C.W.; Formal analysis, X.Z. and J.C.; Investigation, X.Z.; Resources, G.D. and C.W.; Data curation, J.C.; Writing—original draft, X.Z.; Writing—review & editing, J.C. and T.L.; Supervision, G.D. and C.W.; Project administration, X.Z.; Funding acquisition, X.Z. and C.W. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was supported by National Science Foundation of China under grant 61902156 and 62072217.

**Data Availability Statement:** Not applicable.

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

## References

1. Aboushosha, B.; Ramadan, R.A.; Dwivedi, A.D.; El-Sayed, A.; Dessouky, M.M. SLIM: A lightweight block cipher for internet of health things. *IEEE Access* **2020**, *8*, 203747–203757. [CrossRef]
2. Kumar, P.; Kumar, R.; Kumar, A.; Franklin, A.A.; Jolfaei, A. Blockchain and Deep Learning Empowered Secure Data Sharing Framework for Softwarized UAVs. In Proceedings of the 2022 IEEE International Conference on Communications Workshops (ICC Workshops), Seoul, Republic of Korea, 16–20 May 2022; pp. 770–775. [CrossRef]
3. Aljuhani, A.; Kumar, P.; Kumar, R.; Jolfaei, A.; Islam, A.N. Fog intelligence for secure smart villages: Architecture, and future challenges. *IEEE Consum. Electron. Mag.* **2022**, 1–9. Available online: https://ieeexplore.ieee.org/document/9837392 (accessed on 4 January 2023). [CrossRef]
4. Tzounis, A.; Katsoulas, N.; Bartzanas, T.; Kittas, C. Internet of Things in agriculture, recent advances and future challenges. *Biosyst. Eng.* **2017**, *164*, 31–48. [CrossRef]
5. Prvulović, P.; Radosavljević, N.; Babić, D. Analysis of Lightweight Cryptographic Protocols in Precision Agriculture-A Case Study. In Proceedings of the 2021 15th International Conference on Advanced Technologies, Systems and Services in Telecommunications (TELSIKS), Nis, Serbia, 20–22 October 2021; pp. 295–298.
6. Liu, B.T.; Li, L.; Wu, R.X.; Xie, M.M.; Li, Q.P. Loong: A family of involutional lightweight block cipher based on SPN structure. *IEEE Access* **2019**, *7*, 136023–136035. [CrossRef]
7. Bhonge, H.N.; Ambat, M.K.; Chandavarkar, B. An Experimental Evaluation of SHA-512 for Different Modes of Operation. In Proceedings of the 2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT), IEEE, Kharagpur, India, 1–3 July 2020; pp. 1–6.
8. McGrew, D.; Viega, J. Arbitrary block length (ABL) mode: Security without data expansion. *Submiss. IEEE SISWG* **2004**. Available online: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6ff1a8bfcd7a214e1cf2c9c501e724ef753937e1 (accessed on 4 January 2023).
9. Dutta, A.; Nandi, M. Tweakable HCTR: A BBB secure tweakable enciphering scheme. In *International Conference on Cryptology in India*; Springer: Berlin, Germany, 2018; pp. 47–69.
10. McGrew, D.A.; Fluhrer, S.R. The extended codebook (XCB) mode of operation. *Cryptol. Eprint Arch.* **2004**. Available online: http://eprint.iacr.org/2004/278 (accessed on 4 January 2023).
11. Salman, R.S.; Farhan, A.K.; Shakir, A. Lightweight Modifications in the Advanced Encryption Standard (AES) for IoT Applications: A Comparative Survey. In Proceedings of the 2022 International Conference on Computer Science and Software Engineering (CSASE), Hiroshima, Japan, 26–30 March 2022; pp. 325–330.
12. Shibutani, K.; Isobe, T.; Hiwatari, H.; Mitsuda, A.; Akishita, T.; Shirai, T. Piccolo: An ultra-lightweight blockcipher. In *International Workshop on Cryptographic Hardware and Embedded Systems*; Springer: Berlin, Germany, 2011; pp. 342–357.
13. Qi, M.; Chen, J. Secure authenticated key exchange for WSNs in IoT applications. *J. Supercomput.* **2021**, *77*, 13897–13910. [CrossRef]
14. Iqbal, U.; Shafi, S. A provable and secure key exchange protocol based on the elliptical curve diffe–hellman for wsn. In *Advances in Big Data and Cloud Computing*; Springer: Berlin, Germany, 2019; pp. 363–372.
15. Saqib, N. Key exchange protocol for WSN resilient against man in the middle attack. In Proceedings of the 2016 IEEE International Conference on Advances in Computer Applications (ICACA), Coimbatore, India, 24 October 2016; pp. 265–269.
16. Kumar, M. Security of XCB and HCTR. Ph.D. Thesis, Indian Statistical Institute, Kolkata, India, 2018.
17. Chakraborty, D.; Sarkar, P. HCH: A new tweakable enciphering scheme using the hash-counter-hash approach. *IEEE Trans. Inf. Theory* **2008**, *54*, 1683–1699. [CrossRef]
18. Ashoka Raja, Y.; Shah, P.K.; Darji, A.D. FPGA-Based Display Driver Design with Remote Terminal Unit (RTU) Support Using MODBUS. In *Recent Trends in Electronics and Communication*; Springer: Berlin, Germany, 2022; pp. 511–524.
19. Li, J.; Gao, M.; Liu, B.; Cai, Y. Forecasting Aided Distribution Network State Estimation Using Mixed $\mu$PMU-RTU Measurements. *IEEE Syst. J.* **2022**, *16*, 6524–6534. [CrossRef]
20. Uma, U.U.; Ekwue, A.; Nmadu, D.; Eli-Chukwu, N.C. Adaptive distance protection scheme setting in presence of SVC using remote terminal unit. *J. Electr. Eng. Technol.* **2021**, *16*, 1867–1877. [CrossRef]

21. Salahudin, F.; Sudjadi; Setiyono, B. Design of Remote Terminal Unit (RTU) Panel Supply Monitoring Based on IOT Case Study at PLN. In Proceedings of the 2019 sixth International Conference on Information Technology, Computer and Electrical Engineering (ICITACEE), Semarang, Indonesia, 26–27 September 2019; pp. 1–6.
22. Appleby, A. Smhasher & Murmurhash (2012). 2015. Available online: http://code.google.com/p/smhasher (accessed on 4 January 2023).
23. Alvarez, R.; Caballero-Gil, C.; Santonja, J.; Zamora, A. Algorithms for lightweight key exchange. *Sensors* **2017**, *17*, 1517. [CrossRef] [PubMed]