On the Efficiency of Software Implementations of Lightweight Block Ciphers from the Perspective of Programming Languages *†

Abdur Rehman Raza‡, Khawir Mahmood§, Muhammad Faisal Amjad, Haider Abbas, and Mehreen Afzal

National University of Sciences and Technology, Islamabad, Pakistan

Abstract

Lightweight block ciphers are primarily designed for resource constrained devices. However, due to service requirements of large-scale IoT networks and systems, the need for efficient software implementations can not be ruled out. A number of studies have compared software implementations of different lightweight block ciphers on a specific platform but to the best of our knowledge, this is the first attempt to benchmark various software implementations of a single lightweight block cipher across different programming languages and platforms in the cloud architecture. In this paper, we defined six lookup-table based software implementations for lightweight block ciphers with their characteristics ranging from memory to throughput optimized variants. We carried out a thorough analysis of the two costs associated with each implementation (memory and operations) and discussed possible trade-offs in detail. We coded all six types of implementations for three key settings *(64, 80, 128 bits)* of LED (a lightweight block cipher) in four programming languages *(Java, C#, C++, Python)*. We highlighted the impact of choice relating to implementation type, programming language, and platform by benchmarking the seventy-two implementations for throughput and software efficiency on 32 & 64-bit platforms for two major operating systems (Windows & Linux) on Amazon Web Services Cloud. The results showed that these choices can affect the efficiency of a cryptographic primitive by a factor as high as 400.

Keywords: lightweight block-cipher, software implementation, lookup table, LED, IoT, aws EC2

*The published article is available at https://doi.org/10.1016/j.future.2019.09.058

‡Corresponding author: abraza@mcs.edu.pk (A.R. Raza)

§khawir@mcs.edu.pk (K. Mahmood), faisal@nust.edu.pk (M.F. Amjad), haider@mcs.edu.pk (H. Abbas), mehreenafzal@mcs.edu.pk (M. Afzal)

1 Introduction

The proliferation of IoT devices, ranging from personalized fitness companions to smart home sensors is gradually transforming every aspect of human life in fundamental and diverse ways. Furthermore, the amalgamation of IoT with cloud technology and big data is bringing together physical, industrial and biological worlds [56]. According to a prediction by Gartner, more than 20 billion IoT devices will be connected to the Internet by the year 2020 [39]. These IoT devices are constantly producing huge volumes of data which is shared between devices for collaboration and forming ubiquitous systems. The networks of IoT devices necessitate central processing for state-of-the-art intelligent services such as analytics, mining, and prediction. This requirement is met by integrating IoT devices with cloud-based technology resulting in a scalable, robust and highly available collaboration which entails huge potentials and benefits at the individual, society as well as global levels. A key concern regarding the IoT devices is the nature of data accessed and shared by these devices with one another and over cloud infrastructure. This data often includes sensitive personal and mission-critical information for which the most significant factors are privacy and security. Lack of privacy and security diminishes the efficacy of IoT. This, in turn, acts as the primary barrier which needs to be provably surpassed for practical utilization of IoT.

The peculiar cloud-based IoT ecosystem compounds the privacy and security requirements. A balanced approach is sought that deals with resource-constrained IoT devices at one end and performance requirement for a large number of simultaneous cloud-connected devices at the other. Existing standards of National Institute of Standards and Technology(NIST) for encryption (AES [21]) and hash functions (SHA-III [14]) can not be efficiently implemented in resource constrained environments. Therefore, the more suitable options for these tiny IoT devices are low cost, lightweight cryptographic primitives such as block ciphers, stream ciphers, hash functions, and Message Authentication Codes(MAC) [59, 23, 4, 58].

Over the past decade a number of lightweight block ciphers have been designed such as HIGHT [38], KLEIN [31], LED [34], MIBS [40], SPARX [25] and SKINNY [11]. The two block ciphers CLEFIA [60] and PRESENT [18] form part of ISO standard for lightweight block ciphers [29]. Mostly the lightweight block ciphers are designed to support compact hardware implementation in terms of gate count and power consumption. However, FeW [44], ITUbee [42], Robin and Fantomas [32] are also suitable for implementation in software based platforms. For software implementations, the goal is to reduce the memory requirement and increase throughput. Various designs support additional constraints such as low latency, masked implementation and support for both encryption and decryption with minimal overhead [30, 19, 54]. Few designs have been proposed which improve upon or combine the ideas of existing lightweight block ciphers like for example SIMECK [62] combines the best features of two ciphers Simon and Speck [10]. I-PRESENT [63] is an involutive design based on PRESENT [18]. The involution part is inspired from block cipher PRINCE [19] and encryption is identical to decryption except the round keys are used in reverse order.

The lightweight block ciphers perform remarkably well in resource constrained hardware and software platforms but there exists a need for good performance over high-end software machines too. Consider a practical scenario where hundreds of tiny IoT devices are connected to a server. Each device sends a small amount of data *(one block or few bits)* after a specific interval and shares a unique symmetric key with the server for communi-

cating securely. The server needs to decrypt the data received from connected devices in real-time in order to perform some analysis or update the user dashboard. This necessitates the need for efficient software implementation so as not to burden server resources with performing heavy cryptographic operations only.

Motivation. The available software based implementation techniques for lightweight block ciphers include lookup-table based, bit-sliced and use of Single Instruction Multiple Data (SIMD) instructions. The lookup-table based implementation is done by precomputing the small chunks of data, then selecting and aggregating it at runtime. The bit-slice technique introduced in [15], implements the block cipher without lookup tables. It involves breaking down the block cipher into logical bit operations in order to perform N parallel encryptions on an N-bit microprocessor [53]. The use of SIMD instructions for accelerating the AES was presented in [35]. Precisely, the vector permute (`vperm`) instruction is used to perform parallel table lookups in order to increase the throughput. This technique has later been applied to various block ciphers for accelerated implementations and resistance against side channel attacks [49, 51, 45, 61]. The bit-slice and SIMD instruction based implementations deliver very good performance and therefore seem to be the obvious choice for implementation in cloud-based services. However, it has its limitations in practical scenario (explained in previous paragraph) due to the following pitfalls:

- The bit-slice implementation works on N blocks in parallel. Thus it needs N blocks of data to be present on the server before they can be packed together in bundles to perform encryption or decryption. However, if IoT devices are sending small data packets after considerable intervals, then the server has to either wait for the arrival of more data packets or decrypt the single block via bit-slice technique. The first option impairs the server's ability to decrypt the data in real-time whereas the second option gives a performance hit in terms of throughput. In [12], the authors showed that bit-slice implementation is not suitable for scenarios where each message consists of a small number of blocks.

- Although idea of implementing by bit-slice technique has been extended to many lightweight block ciphers but only some are specifically designed for it (Gift [6], PRIDE [2], RECTANGLE [64], RoadRunner [9]). Thus not all the lightweight block ciphers will have similar performance gains from bit-slice implementation. Bitslice implementations of various block ciphers are compared in [7]. The results show that RECTANGLE (whose design took bitslice implementation into consideration) performs remarkably better than others which were not specifically designed for bit-slice technique.

- Not all the higher-level programming languages provide direct support for SIMD instructions, whereas web-services and cloud applications are mostly written in these languages. This limits the usability scenarios of SIMD instructions for fast implementation of cryptographic primitives in higher-level languages.

In the light of the above discussion, look-up table based implementations seem practically more feasible and best suited for cloud-based IoT scenario explained above. Despite having a larger memory footprint in terms of pre-computed lookup tables, they have better performance and large-scale applicability relating to platform and language support.

Our Contributions. Apropos to the proceeding motivation, we contribute in tangible terms to the software based implementations of lightweight block ciphers. The

parameters affecting the performance of a lightweight block cipher in a cloud-based IoT network include the choice of the lightweight block cipher, implementation type, programming language, operating system, and architecture. There exist a considerable body of literature that compares particular implementation of various lightweight block ciphers on a specific/single platform [12, 47, 43, 27, 24, 20], but to the best of our knowledge, this is the first attempt to benchmark lookup table based software implementations of a single lightweight block cipher across different programming languages and platforms using Amazon Web Services (AWS) cloud architecture. Specifically, we have made the following contributions in this paper:

- We defined six lookup-table based software implementations for lightweight block ciphers with their characteristics and possible trade-offs, ranging from memory to throughput optimized variants.
- We discussed the two types of costs (memory and operations) associated with each implementation and shed light on the efficient computation of lookup tables and round operations.
- We elaborated upon packing and unpacking cost associated with each implementation and explained the efficient conversion of plaintext and key bytes into the format and data type required by each implementation.
- We implemented Light Encryption Device (LED) block cipher for three key settings *(64, 80, 128 bits)* in four programming languages *(Java, C#, C++, Python)* for all six implementation types.
- We benchmarked the seventy-two implementations for throughput and software efficiency on 32 & 64-bit platform for two major Operating Systems (Windows & Linux) on Amazon Web Services (AWS) Cloud. The results show the amount of impact *the choice of implementation type, programming language and platform* has on the efficiency of a cryptographic primitive.

The rest of this paper is organized as follows: In Section 2 we provide a comprehensive breakdown of LED specifications. This is followed by a detailed description and trade-offs related to six LookUp Table based (LUT) software implementations (4-bit Serial, 4-bit LUT, 8-bit LUT, 16-bit LUT, 32-bit LUT, and 64-bit LUT) of LED block cipher in Section 3. The Section 4 presents performance results of all implementations. A comprehensive account of related & future work is given in Section 5 and Section 6 concludes the paper.

The implementation codes are available at
https://github.com/rzpbcodes/LightWeightBlockCiphers

2 LED Block Cipher

LED is a lightweight block cipher with operations similar to AES [21] like `sbox`, `shiftRows` and `mixColumns`. The cipher supports 64-bit block length and key lengths of 64 to 128 bits in multiples of 4. LED block cipher does not employ any key schedule, rather the user provided master key is used as-is where required. Moreover, the round key is mixed into the plaintext after every four rounds, called `step`. This helps in realizing compact hardware implementation while enabling the provision of concrete security

bounds under the related key attacks. Although the non-existence of key-schedule seems dangerous and makes the cipher vulnerable to different types of attacks [16, 17], special care has been taken in the design of LED to thwart against this e.g. resistance to slide-attacks [34].

The 64-bit plaintext block p is conceptually arranged in a 4×4 matrix of 16 nibbles(4-bit) as

$$\begin{bmatrix} p_0 & p_1 & p_2 & p_3 \\ p_4 & p_5 & p_6 & p_7 \\ p_8 & p_9 & p_{10} & p_{11} \\ p_{12} & p_{13} & p_{14} & p_{15} \end{bmatrix}$$

Each nibble is an element of $GF(2^4)$ with underlying polynomial for finite field multiplication as $X^4 \oplus X \oplus 1$. The elements $p_0|p_1|p_2|\cdots|p_{14}|p_{15}$ are loaded row-wise into the *state* matrix. This is more hardware friendly as compared to loading the *state* column-wise as is the case in AES [50].

The master key K consists of l nibbles $(k_0|\ k_1|\ k_2|\ \cdots|\ k_{l-2}|\ k_{l-1})$. All the subkeys are conceptually arranged as a 4×4 square matrix just like the input *state* matrix. For 64-bit master key, there is only one subkey sk^0 which is equal to the master key K. For 128-bit master key, the two subkeys sk^0 and sk^1 are equal to left and right half of the master key $K = sk^0|sk^1$. For master key with $64 < length < 128$. (in multiple of 4), the first subkey sk^0 consists of first 16 nibbles of the master key K and the second subkey sk^1 is computed as

$$sk_i^1 = K_{(i+16) \mod l} \quad i : 0 \rightarrow 15.$$

The 4×4 subkey matrices for 64, 80 and 128 bit master keys K are as

$$\begin{bmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \\ k_{12} & k_{13} & k_{14} & k_{15} \end{bmatrix}$$

subkey for 64-bit master key.

$$\begin{bmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \\ k_{12} & k_{13} & k_{14} & k_{15} \end{bmatrix} \begin{bmatrix} k_{16} & k_{17} & k_{18} & k_{19} \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \end{bmatrix}$$

subkeys for 80-bit master key.

$$\begin{bmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \\ k_{12} & k_{13} & k_{14} & k_{15} \end{bmatrix} \begin{bmatrix} k_{16} & k_{17} & k_{18} & k_{19} \\ k_{20} & k_{21} & k_{22} & k_{23} \\ k_{24} & k_{25} & k_{26} & k_{27} \\ k_{28} & k_{29} & k_{30} & k_{31} \end{bmatrix}$$

subkeys for 128-bit master key.

Figure 1 illustrates the encryption operation of the LED block cipher which primarily consists of mixing the subkeys into *state* and performing `step` operation.

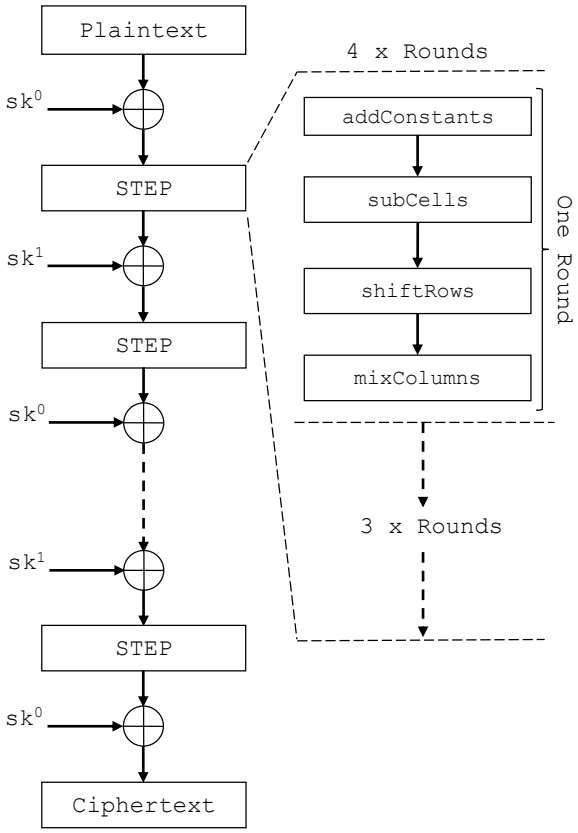

Figure 1: Encryption operation of LED block cipher.

Pseudo-code for LED block cipher encryption method is shown in Listing 1. The `addRoundKey(state, sk)` method mixes the `subkey` into *state* using binary Exclusive-Or (xor) operation ($\oplus$). For 64-bit master key, both subkeys sk^0 and sk^1 are equal to the master key K. The `step(state)` method updates the *state* by applying four rounds. Each round consists of four operations, `addConstants`, `subCells`, `shiftRows` and `mixColumns` as shown in Figure 2. The number of steps to be performed depends on the key length. For 64-bit master key, the number of steps is 8 and for the master key of length $64 < length \leq 128$ number of steps is equal to 12. A shows the test vectors and performance results of the LED block cipher.

```
for i=0 to s-1 do {
addRoundKey(state, sk^0);
step(state);
addRoundKey(state, sk^1);
step(state);
}
addRoundKey(state, sk^0);
```

Listing 1: Pseudo-code for LED Block Cipher Encryption.

`addConstants`. Each round mixes 8 bits of key size constant and 6 bits of Linear Feedback Shift Register (LFSR) constant by xor operation with the first and second column of the *state* matrix. The key length of the master key K is expressed as key size constant of 8 bits (ks_7, ks_6, ..., ks_0) where ks_7 is Most Significant bit(MSb). The 6 bits of LFSR constant rc_5, rc_4, rc_3, rc_2, rc_1, rc_0, are computed as shown in Figure 3. All six state bits of the LFSR are initialized to zero at start and values are updated before using

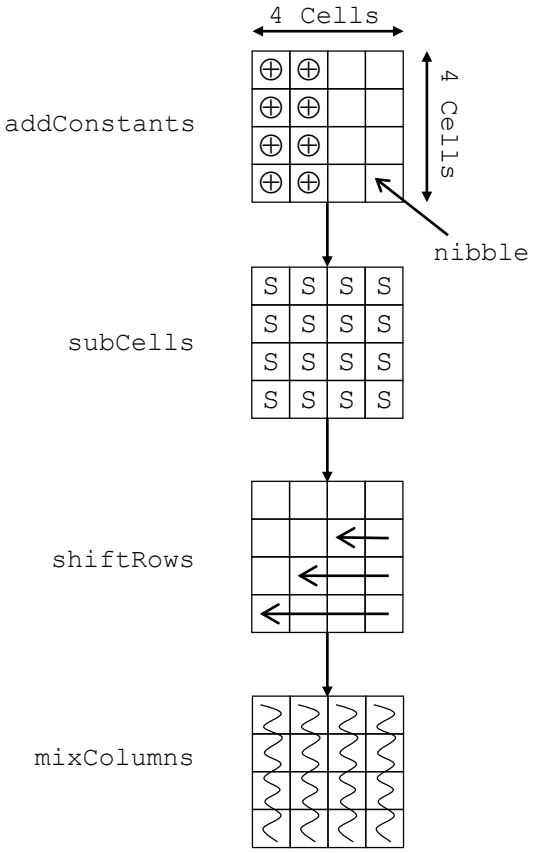

Figure 2: One round of LED block cipher.

in the round. To update, the LFSR is shifted one position left and rc_0 is updated with a new value computed as $rc_5 \oplus rc_4 \oplus 1$. Table 1 shows round constant values in hexadecimal notation. Matrix representation of LFSR and key size constant bits which are added in *state* matrix by xor operation is as

$$\begin{bmatrix} 0 \oplus (ks_7|ks_6|ks_5|ks_4) & rc_5|rc_4|rc_3 & 0 & 0 \\ 1 \oplus (ks_7|ks_6|ks_5|ks_4) & rc_2|rc_1|rc_0 & 0 & 0 \\ 2 \oplus (ks_3|ks_2|ks_1|ks_0) & rc_5|rc_4|rc_3 & 0 & 0 \\ 3 \oplus (ks_3|ks_2|ks_1|ks_0) & rc_2|rc_1|rc_0 & 0 & 0 \end{bmatrix}$$

`subCells`. The sixteen nibbles of the *state* are updated with sbox values. LED uses sbox of PRESENT [18] block cipher as given in Table 2.

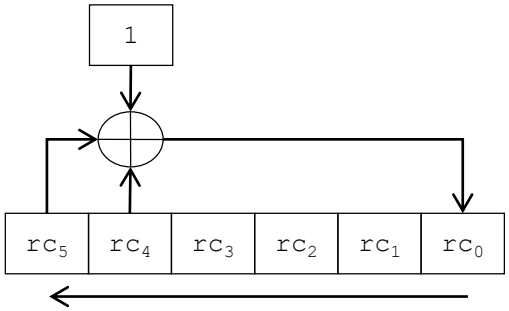

Figure 3: Generation of LED round constants by LFSR.

Table 1: Round constants for LED block cipher in Hex notation.

Rounds	Constants
1 - 10	01,03,07,0F,1F,3E,3D,3B,37,2F
11 - 20	1E,3C,39,33,27,0E,1D,3A,35,2B
21 - 30	16,2C,18,30,21,02,05,0B,17,2E
31 - 40	1C,38,31,23,06,0D,1B,36,2D,1A
41 - 48	34,29,12,24,08,11,22,04

Table 2: Substituion-box (sbox) of PRESENT block cipher.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$sbox[x]$	c	5	6	b	9	0	a	d	3	e	f	8	4	7	1	2

`shiftRows`. The shift rows operation is performed similarly as in AES. The i^{th} row of the *state* matrix is left rotated cyclically for i number of cell where $i = 0, 1, 2, 3$.

$$\begin{bmatrix} p_0 & p_1 & p_2 & p_3 \\ p_4 & p_5 & p_6 & p_7 \\ p_8 & p_9 & p_{10} & p_{11} \\ p_{12} & p_{13} & p_{14} & p_{15} \end{bmatrix} \rightarrow \begin{bmatrix} p_0 & p_1 & p_2 & p_3 \\ p_5 & p_6 & p_7 & p_4 \\ p_{10} & p_{11} & p_8 & p_9 \\ p_{15} & p_{12} & p_{13} & p_{14} \end{bmatrix}$$

`mixColumns`. Each column of the *state* is multiplied by a 4×4 MDS matrix M. The MDS matrix M is derived from a simple 4×4 matrix A such that $A^4 = M$. The matrix A can efficiently be realized in hardware for very compact serial implementation. It has simple elements $0, 1, 2$ and 4 and there is no memory element or control logic involved in storing the temporary multiplication results [33].

$$A^4 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 4 & 1 & 2 & 2 \end{bmatrix}^4 = \begin{bmatrix} 4 & 1 & 2 & 2 \\ 8 & 6 & 5 & 6 \\ b & e & a & 9 \\ 2 & 2 & f & b \end{bmatrix} = M$$

3 Software Implementations

In the following subsections, we define six lookup-table based software implementations for lightweight block ciphers, ranging from memory to speed optimized variants. We explain the implementations for LED lightweight block cipher because it is a Substitution Permutation Network (SPN) with architecture similar to AES [21]. AES is NIST standard for block ciphers and in fact, the most extensively studied design. Its wide trail strategy provides concrete security bounds against differential and linear cryptanalysis [22]. Over the years, many lightweight block ciphers including LED have been designed with a structure similar to AES such as KLEIN [31], Midori [5], Mysterion [41], Skinny [11], and Zorro [30]. Therefore, explaining implementation techniques for LED shall help in covering a large range of lightweight block ciphers to which these techniques can be easily extended. In addition to it, LED employs recursive MDS matrix in permutation layer which helps in realizing the 4-bit Serial implementation. Showing the real essence

of serial implementation may have not been possible if some other SPN block cipher would have been selected. Moreover, if we had chosen some 64-bit Feistel lightweight block cipher then it would have not been possible to explain 64-bit LUT implementation as it's round function would be operating on 32-bits. There are two types of costs associated with each implementation i.e memory and operations. Memory requirement consists of precomputed tables like `sbox`, `roundConstants` and multiplication tables whereas operations' cost is the number of basic operations (`xor`, `and`, `shift`) required to perform one round or encryption of complete block. We explain both the costs for each implementation type in detail and discuss possible trade-offs in terms of memory requirement and number of operations.

3.1 4-bit Serial

The 64-bit *state* and subkey sk^0 are stored in two byte arrays of length 16 each. The 4-bit sbox and 32 LFSR constants as shown in Table 2 are stored in two byte arrays of length 16 and 32 respectively. The round key sk^0 is added to the *state* matrix by sixteen xor operations i.e adding each nibble of the round key to the corresponding nibble of the *state* matrix. Byte value of LFSR constant for each round is fetched from the LUT and split into two bytes x, y such that $x = |rc_5|rc_4|rc_3$ and $y = rc_2|rc_1|rc_0$. These two bytes x, y are then added into nibbles $state_1$, $state_9$ and $state_5$, $state_{13}$ of the *state* matrix respectively. Relevant bits of key size constant and the values `0x00`, `0x01`, `0x02` and `0x03` are mixed by xor and stored in four bytes. These four bytes are then added to the nibbles $state_0$, $state_4$, $state_8$, $state_{12}$ of the *state* matrix in each round. In total, the `addConstants` operation is performed by eight `xor`, two `and`, one `shift` and two `lookup` operations (C - Listing 8).

The `subCells` operation updates the *state* by sixteen lookups from the sbox. The `shiftRows` operation needs another sixteen lookups from the current *state* matrix and stores the values at shifted indices in the new *state* matrix. However, the `subCells` and `shiftRows` operations can be combined together. In this case, the updated values of nibbles after lookup from the sbox are directly stored in the new *state* matrix at appropriate indices with respect to `shiftRows` operation. This reduces the extra lookups and both operations can be completed in just sixteen lookups as given in C - Listing 9.

The `mixColumns` operation is implemented using the 4×4 matrix A. Each column of the *state* matrix is multiplied with the matrix A for four times. This is equal to one-time multiplication of *state* matrix columns by matrix M. Matrix A consists of only four distinct elements i.e 0, 1, 2, 4. Multiplication with 0 and 1 is straight forward. Multiplying an element q of $GF(2^4)$ with 2 is similar to multiplying the polynomial representation of element q with x modulo the polynomial $X^4 \oplus X \oplus 1$ (details in B). Thus `xTimes` is achieved by left shift of one bit position and a conditional xor with underlying field polynomial (see Listing 2). Multiplication with 4 is equal to multiplying two times with 2 i.e `x2Times(q) = xTimes(xTimes(q))`. Thus, multiplying one column of *state* with matrix A for four times is implemented by 12 `xor` and 16 calls to the method `xTimes` (C - Listing 10).

The 4-bit Serial implementation requires 52 bytes of memory for sbox, LFSR constants and key size constants. The complete encryption of one block consists of 1936 `xor`, 64 `and`, 32 `shifts`, 576 `lookup` operations and 2048 calls to `xTimes` method.

Trade-Offs. The 32 byte LFSR constants LUT can be omitted and these can be computed on the fly by implementing LFSR. This will reduce the memory cost at the

```
byte xTimes(byte q) {
q <<= 1;
if ((q & 0x10) == 0x10)
q ^= poly;
return (q & 0x0F)
}
```

Listing 2: xTimes - Multiplying q by 2 in $GF(2^4)$.

expense of extra operations. Each round will need to process an extra 4 xor and 3 shift operations for updating the LFSR state. On the other hand, the LFSR constants for byte values x, y ($x = |rc_5|rc_4|rc_3$ and $y = rc_2|rc_1|rc_0$) can be precomputed and stored in two different byte arrays of length 32 each. This will reduce the number of operations required to split values of LFSR constants at runtime but require extra 32 bytes of memory. In another possible case, the multiplication by 2 and 4 can be precomputed and stored in two byte arrays of length 16. Then multiplication in `mixColumns` operation can be carried out by lookups instead of calls to `xTimes` method.

3.2 4-bit LUT

The 64-bit *state* and subkey sk^0 are stored in two byte arrays of length 16 each. A byte array of length 16 is used to store the values of 4-bit sbox. The round key sk^0 is mixed into the *state* matrix by sixteen xor operations. The LFSR constants are split into three MSb & Least Significant bits (LSb) and stored in two different byte arrays, each of length 16. Then, the pre-splitted value is fetched from appropriate LUT and added to the $state_1$, $state_9$ or $state_5$, $state_{13}$ by xor operation. This reduces the number of operations for computing LFSR constant value but increases memory requirement by 16 bytes. Relevant bits of key size constant and the values `0x00`, `0x01`, `0x02` and `0x03` are added together by xor operation and stored in four bytes. These four bytes are then added to $state_0$, $state_4$, $state_8$ and $state_{12}$ respectively in each round. The complete `addConstants` operation consists of four lookups and eight xor operations.

The `mixColumns` operation is performed by multiplying with the matrix M instead of matrix A. Other than 1, there are ten distinct elements in the matrix M (see `mixColumns` operation in Section 2). The multiplication of each element q of $GF(2^4)$ with each distinct element m of matrix M is precomputed and stored in multiplication tables ($mulTable_m$). The size of each multiplication table is 16 bytes. This increases the memory cost by 160 bytes but reduces the number of operations required to be performed for matrix multiplication. The value at index i in a $mulTable_m$ is computed by multiplying m with $sbox[i]$ in $GF(2^4)$.

$$mulTable_m[i] = m \times sbox[i] \quad i : 0 \rightarrow 15.$$

The entries of $mulTable_2$ are shown in Table 3. Computing multiplication tables by combining the effect of *sbox* and finite field multiplication help in performing all three operations `subCells`, `shiftRows` and `mixColumns` together. This reduces the overall number of basic operations required to implement one round. All subsequent implementations combine the above mentioned three operations together. Listing 3 shows the combined implementation of `subCells`, `shiftRows` and `mixColumns` operations.

Table 3: Multiplication Table for element 2 ($mulTable_2$) in $GF(2^4)$ for combined subCells & mixColumns Operations.

i	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$2 \times sbox[i]$	b	a	c	5	1	0	7	9	6	f	d	3	8	e	2	4

```
void SubCellShiftRowMixColumns(byte[] state)   {
byte[] temp = new byte[16];

temp[0] = (mul4[state[0]] ^ sbox[state[5]] ^ mul2[state[10]] ^ mul2[state
    [15]]);
temp[4] = (mul8[state[0]] ^ mul6[state[5]] ^ mul5[state[10]] ^ mul6[state
    [15]]);
temp[8] = (mulb[state[0]] ^ mule[state[5]] ^ mula[state[10]] ^ mul9[state
    [15]]);
temp[12] = (mul2[state[0]] ^ mul2[state[5]] ^ mulf[state[10]] ^ mulb[state
    [15]]);
⋮
state = temp;
}
```

Listing 3: Combined 4-bit LUT Implementation of subCells, shiftRows & mixColumns Operations.

The 4-bit LUT implementation requires 244 bytes of memory for sbox, LFSR constants, multiplication tables and key size constants. The complete encryption of one block consists of 1936 `xor` and 2112 `lookup` operations.

Trade-Offs. Multiplication tables for only five distinct elements (2, 4, 8, b, e) of matrix M are precomputed and stored in byte arrays. The multiplication values for remaining elements of matrix M can be computed at runtime by

$$
\begin{aligned}
mul_5[i] &= mul_4[i] \oplus sbox[i];\\
mul_6[i] &= mul_2[i] \oplus mul_4[i];\\
mul_9[i] &= mul_8[i] \oplus sbox[i];\\
mul_a[i] &= mul_b[i] \oplus sbox[i];\\
mul_f[i] &= mul_e[i] \oplus sbox[i];
\end{aligned}
$$

This reduces the memory requirement by 80 bytes, but it needs extra 20 lookups and 20 xor operations for matrix multiplication in each round.

3.3 8-bit LUT

The 64-bit *state* and subkey sk^0 are stored in two byte arrays of length 8. The two consecutive nibbles p_{2i} and p_{2i+1} of *state* matrix are concatenated together to form a byte b_i as

$p_{15}\|p_{14}$	$\cdots$	$\cdots$	$p_3\|p_2$	$p_1\|p_0$
b_7	$\cdots$	$\cdots$	b_1	b_0

$$\begin{bmatrix} p_0 & p_1 & p_2 & p_3 \\ p_4 & p_5 & p_6 & p_7 \\ p_8 & p_9 & p_{10} & p_{11} \\ p_{12} & p_{13} & p_{14} & p_{15} \end{bmatrix} \longrightarrow \begin{bmatrix} b_0 & b_1 \\ b_2 & b_3 \\ b_4 & b_5 \\ b_6 & b_7 \end{bmatrix}$$

The values of the sbox are stored in a byte array of length 256. This large $sbox8$ has input and output of 8-bits. The value at index b of the $sbox8$ is computed as

$$sbox8[b] = sbox[b_{msb(4)}] \mid sbox[b_{lsb(4)}]; \quad b : 0 \rightarrow 255$$

Now the complete *state* can be updated by its sbox values in 8 lookups as two nibbles are updated in a single lookup. This reduces the number of lookups by a factor of 2 in each round. However, this 8-bit sbox needs 240 more bytes of memory than the 4-bit sbox. The round key sk^0 is mixed into the *state* matrix by 8 xor operations. Relevant bits of LFSR constants, key size constant and the values `0x00`, `0x01`, `0x02` and `0x03` are concatenated together to form byte values.

$$\begin{aligned} b_0 &= (00 \oplus (ks_7|ks_6|ks_5|ks_4)) \mid (rc_5|rc_4|rc_3) \\ b_2 &= (01 \oplus (ks_7|ks_6|ks_5|ks_4)) \mid (rc_2|rc_1|rc_0) \\ b_4 &= (02 \oplus (ks_3|ks_2|ks_1|ks_0)) \mid (rc_5|rc_4|rc_3) \\ b_6 &= (03 \oplus (ks_3|ks_2|ks_1|ks_0)) \mid (rc_2|rc_1|rc_0) \end{aligned}$$

These byte values are computed for byte positions b_0, b_2, b_4 and b_6 of the *state* matrix for all 32 rounds and stored in four byte arrays of length 32 each. Then `addConstants` operation is performed by four lookups (one from each round constants LUT) and four xor operations.

The `mixColumns` operation is performed similarly as described in Section 3.2 (4-bit LUT) except this 8-bit LUT implementation employs larger multiplication tables ($mulTable_m$). Just like the $sbox8$, these larger multiplication tables have input and output of 8-bits. The values of $mulTable_m$ for each distinct element m of matrix M are precomputed and stored in ten byte arrays of length 256 each. This increases the memory cost from 160 bytes to 2560 bytes but reduces the number of lookups and xor required for implementing `mixColumns` operation. The 8-bit value at index b of $mulTable_m$ is computed as

$$\begin{aligned} mulTable_m[b] &= (m \times sbox[b_{msb(4)}])|(m \times sbox[b_{lsb(4)}]) \\ b &= 0 \rightarrow 255 \end{aligned}$$

Since the adjacent nibbles are concatenated together to form bytes, the `shiftRows` operation is performed as follows.

$$\begin{bmatrix} b_0 & b_1 \\ b_2 & b_3 \\ b_4 & b_5 \\ b_6 & b_7 \end{bmatrix} \xrightarrow{SR} \begin{bmatrix} b_0 & b_1 \\ (b_3 \ll_4)|(b_2 \gg_4) & (b_3 \gg_4)|(b_2 \ll_4) \\ b_5 & b_4 \\ (b_7 \ll_4)|(b_6 \gg_4) & (b_7 \ll_4)|(b_6 \gg_4) \end{bmatrix}$$

First row is not shifted so b_0 and b_1 stays in place. In order to shift the second and fourth row, the two bytes of the row are left and right shifted for four bit positions and then concatenated together to form new byte values. Third row is shifted 2 cell positions,

```
void SubCellShiftRowMixColumns(ref byte[] state)   {
byte[] temp = new byte[8];
byte b2, b3, b6, b7;

b2 = (byte)((state[2] >> 4) ^ (state[3] << 4));
b3 = (byte)((state[2] << 4) ^ (state[3] >> 4));

temp[0] = (mul4[state[0]] ^ sbox[b2] ^ mul2[state[5]] ^ mul2[b6]);
temp[2] = (mul8[state[0]] ^ mul6[b2] ^ mul5[state[5]] ^ mul6[b6]);
temp[4] = (mulb[state[0]] ^ mule[b2] ^ mula[state[5]] ^ mul9[b6]);
temp[6] = (mul2[state[0]] ^ mul2[b2] ^ mulf[state[5]] ^ mulb[b6]);
⋮
state = temp;
}
```

Listing 4: Combined 8-bit LUT Implementation of subCells, shiftRows & mixColumns Operations.

so its simply swap of the bytes b_4 and b_5. Listing 4 shows the combined 8-bit LUT implementation of `subCells`, `shiftRows` and `mixColumn` operations.

The 8-bit LUT implementation requires 2944 bytes of memory for sbox, round constants and multiplication tables. The complete encryption of one block consists of 1096 `xor`, 256 `shifts` and 1152 `lookup` operations.

Trade-Offs. Only the LFSR constants are precomputed and stored in two byte arrays of length 32 each. Relevant bits of the key size constant are mixed with `0x00`, `0x01`, `0x02`, `0x03` and stored separately as four byte values. This reduces the memory cost by 60 bytes, but it requires extra four xor operations to be performed in every round for mixing the key size constant into the *state* matrix. Similar to 4-bit LUT implementation, the multiplication tables for only 5 distinct elements (2, 4, 8, b, e) of matrix M can be stored in five byte arrays. The multiplication values for remaining elements of the matrix M can be computed at runtime. This reduces the memory cost for multiplication tables from 2560 bytes to 1280 bytes. But it will increase the number of lookups and xor operations from 32, 24 to 42, 34 respectively for each round. In another possible way, the memory requirement of 8-bit LUT implementation can be reduced by implementing matrix multiplication in serial way just like 4-bit Serial implementation. The multiplication of two nibbles with 2 and 4 can precomputed and stored in two byte arrays of length 256 each. This will reduce the memory required for multiplication tables from 2560 to 512 bytes at the expense of extra lookups and xor operations.

3.4 16-bit LUT

The 64-bit *state* and subkey sk^0 are stored in two arrays of ushort (16-bit unsigned integer) each of length 4. Nibbles belonging to one column of the *state* matrix are concatenated together to form a ushort word u_i as

$$\begin{array}{c|c} p_{13}|p_9|p_5|p_1 & p_{12}|p_8|p_4|p_0 \\ \hline u_1 & u_0 \end{array}$$

$$\begin{array}{c|c} p_{15}|p_{11}|p_7|p_3 & p_{14}|p_{10}|p_6|p_2 \\ \hline u_3 & u_2 \end{array}$$

This particular arrangement of storing nibbles in ushort words enables efficient implementation of matrix multiplication with one column of the *state* matrix. The round key sk^0 is added to the *state* matrix by four xor operations. The LFSR constant bits for each round are concatenated to form a ushort word and stored in an array of length 32.

$$\begin{aligned} u &= b|a|b|a \\ a &= 0|rc_2|rc_1|rc_0 \\ b &= 0|rc_5|rc_4|rc_3 \end{aligned}$$

Then LFSR constant is added to the word u_1 of the *state* matrix by one lookup and xor operation, in each round. The xor of key size constant with values `0x00`, `0x01`, `0x02` and `0x03` is precomputed and stored as a single ushort word. This key size constant is then added to the word u_0 of the *state* matrix in each round by one xor operation. The complete `addConstants` operation consists of two xor and one lookup.

The 16-bit LUT implementation employs four lookup tables (T_0, T_1, T_2, T_3) of 4-bit input and 16-bit output. Each lookup table T_i takes input of *ith* 4-bit nibble of 16-bit string $(U = U_3|U_2|U_1|U_0)$ and outputs multiplication of nibble U_i with all four elements of the *ith* column of matrix M. The 16-bit outputs from all four lookup tables are xor together to produce output of matrix multiplication.

$$M \times U = U' \longrightarrow \begin{bmatrix} 4 & 1 & 2 & 2 \\ 8 & 6 & 5 & 6 \\ b & e & a & 9 \\ 2 & 2 & f & b \end{bmatrix} \times \begin{bmatrix} U_0 \\ U_1 \\ U_2 \\ U_3 \end{bmatrix} = \begin{bmatrix} U'_0 \\ U'_1 \\ U'_2 \\ U'_3 \end{bmatrix}$$

$$\begin{aligned} & T_0[U_0] = 2.U_0|b.U_0|8.U_0|4.U_0 \\ \oplus\ & T_1[U_1] = 2.U_1|e.U_1|6.U_1|1.U_1 \\ \oplus\ & T_2[U_2] = f.U_2|a.U_2|5.U_2|2.U_2 \\ \oplus\ & T_3[U_3] = b.U_3|9.U_3|6.U_3|2.U_3 \\ & U' = \ U'_3 \mid U'_2 \ \mid U'_1 \mid U'_0 \end{aligned}$$

The computation of `mixColumns` operation for one column of the *state* matrix is completed by four `lookups`, three `xor`, three `shift` and four `and` operations. In actual, the four lookup tables are computed after taking into account the values from `subCells` operation. This helps in combining the `subCells` and `shiftRow` operations with the matrix multiplication as shown in Listing 5.

The 16-bit LUT implementation requires 194 bytes of memory for round constants and four lookup tables. The complete encryption of one block consists of 484 `xor`, 512 `and`, 384 `shift` and 544 `lookup` operations.

3.5 32-bit LUT

The 32-bit LUT implementation is a combination of 8-bit LUT and 16-bit LUT implementations. The bytes are formed from nibbles as in 8-bit LUT and lookup tables are computed similar to 16-bit LUT implementation excepts now its for 8-bit input values. The 64-bit *state* and subkey sk^0 are stored in two arrays of uint (32-bit unsigned integer),

```
void SubCellShiftRowMixColumns(UInt16[] state)   {
UInt16[] temp = new UInt16[4];

temp[0]  = T0[(state[0] >> 00) & 0x0F];
temp[0] ^= T1[(state[1] >> 04) & 0x0F];
temp[0] ^= T2[(state[2] >> 08) & 0x0F];
temp[0] ^= T3[(state[3] >> 12) & 0x0F];
⋮
state = temp;
}
```

Listing 5: Combined 16-bit LUT Implementation of subCells, shiftRows & mixColumns Operations.

each of length 2. The two consecutive nibbles p_{2i} and p_{2i+1} of *state* matrix are concatenated together to form a byte b_i. Then the bytes belonging to one column of the *state* matrix are concatenated together to form a uint word u_i as

$$\begin{array}{c|c|c|c|c} p_{15}|p_{14} & \cdots & \cdots & p_3|p_2 & p_1|p_0 \\ \hline b_7 & \cdots & \cdots & b_1 & b_0 \end{array}$$

$$\begin{array}{c|c} b_7|b_5|b_3|b_1 & b_6|b_4|b_2|b_0 \\ \hline u_1 & u_0 \end{array}$$

The round key sk^0 is added to the *state* matrix by two xor operations. Relevant bits of the LFSR constants, key size constant and values `0x00`, `0x01`, `0x02` and `0x03` are mixed by xor operation and concatenated together to form a 32-bit round constant word u as

$$\begin{aligned} u &= d|(b \oplus 03)|c|(b \oplus 02)|d|(a \oplus 01)|c|(a \oplus 00) \\ a &= ks_7|ks_6|ks_5|ks_4 \\ b &= ks_3|ks_2|ks_1|ks_0 \\ c &= 0|rc_5|rc_4|rc_3 \\ d &= 0|rc_2|rc_1|rc_0 \end{aligned}$$

The 32-bit round constant words are then precomputed for all rounds and stored in a uint array of length 32. In each round, one word is fetched from round constants array and is added to the word u_0 of the *state* matrix. The complete `addConstants` operation consists of one lookup and one xor.

The 32-bit LUT implementation uses four lookup tables (T_0, T_1, T_2, T_3). Each table has an 8-bit input and 32-bit output. The *ith* entry of the table T_i is computed by splitting the 8-bit input (b) into two nibbles and multiplying each nibble by all four elements of the *ith* column of the matrix M. For example, the contents of the lookup table T_0 are computed as

$$\begin{aligned} T_0[i] &= 2.p_1|2.p_0|b.p_1|b.p_0|8.p_1|8.p_0|4.p_1|4.p_0 \\ i &= i_{msb(4)}|i_{lsb(4)} = p_1|p_0 \\ i &= 0 \rightarrow 255 \end{aligned}$$

```
void SubCellShiftRowMixColumns(uint[] state)   {
uint[] temp = new uint[2];
byte b2, b3, b6, b7;
b2 = (byte)(((state[0] >> 12) & 0xF) ^ ((state[1] >> 4) & 0xF0));
⋮
temp[0] = T0[state[0] & mask];
temp[0] ^= T1[b2];
temp[0] ^= T2[(state[1] >> 16) & mask];
temp[0] ^= T3[b6];
⋮
state = temp;
}
```

Listing 6: Combined 32-bit LUT Implementation of subCells, shiftRows & mixColumns Operations.

The value of nibbles p_0 and p_1 is updated from the sbox before multiplying with elements of *ith* column the matrix M. These lookup tables are precomputed and stored in four uint arrays of length 256. The 32-bit outputs from all four lookup tables are added together by xor operation to produce output of `subCells`, `shiftRows` and `mixColumns` as shown in Listing 6. The `shiftRows` operation over bytes is performed similarly as mentioned in Section 3.3.

The 32-bit LUT implementation requires 4224 bytes of memory for round constants and four lookup tables. The complete encryption of one block consists of 242 `xor`, 384 `and`, 320 `shift` and 288 `lookup` operations.

3.6 64-bit LUT

The 64-bit *state* and subkey sk^0 are stored in two ulong words (64-bit unsigned integer). The two consective nibbles p_{2i} and p_{2i+1} of *state* matrix are concatenated together to form a byte b_i. Then these eight bytes are concatenated together to form a ulong word u as

$p_{15}\|p_{14}$	$\cdots$	$\cdots$	$p_3\|p_2$	$p_1\|p_0$
b_7	$\cdots$	$\cdots$	b_1	b_0
u				

The round key sk^0 is added to the *state* by one xor operation. The round constants for all 32 rounds are precomputed as ulong words and stored in an array of length 32. Each 64-bit word u of the round constants array is computed by concatenating together the values of LFSR constant, key size constant and values `0x00`, `0x01`, `0x02` and `0x03`. The complete `addConstants` operation is then performed by one lookup and one xor.

$$
\begin{aligned}
u &= j|z|(x \oplus 03)|j|y|(x \oplus 02)|j|z|(w \oplus 01)|j|y|(w \oplus 00) \\
w &= ks_7|ks_6|ks_5|ks_4 \quad , \quad y = 0|rc_5|rc_4|rc_3 \\
x &= ks_3|ks_2|ks_1|ks_0 \quad , \quad z = 0|rc_2|rc_1|rc_0 \\
j &= 0|0|0|0|0|0|0|0
\end{aligned}
$$

```
ulong SubCellShiftRowMixColumns(ulong state)    {
ulong temp = 0;

temp = T0[state & 0xFF];
temp ^= T1[state >> 8 & 0xFF];
temp ^= T2[state >> 16 & 0xFF];
temp ^= T3[state >> 24 & 0xFF];
temp ^= T4[state >> 32 & 0xFF];
temp ^= T5[state >> 40 & 0xFF];
temp ^= T6[state >> 48 & 0xFF];
temp ^= T7[state >> 56 & 0xFF];

return temp;
}
```

Listing 7: Combined 64-bit LUT Implementation of subCells, shiftRows & mixColumns Operations.

The 64-bit LUT implementation employs eight lookup tables (T_0, T_1, T_2, ..., T_7). Each lookup table takes an 8-bit input and outputs a 64-bit ulong word. In each lookup table, the updated value of the input from sbox and matrix multiplication is placed at the appropriate position in 64-bit output, keeping in view the shift row operation. The remaining byte positions within the ulong word are set to value `0x00`. For example, the value at index i of the table T_0 and T_1 is computed as

$$
\begin{aligned}
T_0[i] &= j \mid w \mid j \mid x \mid j \mid y \mid j \mid z \\
T_1[i] &= w \mid j \mid x \mid j \mid y \mid j \mid z \mid j \\
w &= 2.p_1|2.p_0, \quad y = 8.p_1|8.p_0 \\
x &= b.p_1|b.p_0, \quad z = 4.p_1|4.p_0 \\
j &= 0|0|0|0|0|0|0|0 \\
i &= i_{msb(4)}|i_{lsb(4)} = p_1|p_0 \\
i &= 0 \rightarrow 255
\end{aligned}
$$

The 64-bit outputs of lookup tables T_0 and T_1 for few indices are provided in D. In order to perform `subCells`, `shiftRows` and `mixColumns`, the *ith* byte of the *state* is input to table T_i and all eight 64-bit outputs are mixed together by xor operation. This is achieved by seven `xor`, eight `and`, seven `shift` and eight `lookup` operations, as shown in Listing 7.

The 64-bit LUT implementation requires 16,640 bytes of memory for round constants and eight lookup tables. The complete encryption of one block consists of 265 `xor`, 256 `and`, 256 `shift` and 288 `lookup` operations.

Trade-Offs. Instead of 8-bit input and 64-bit output, the lookup tables can be computed for 4-bit input and 64-bit output(each corresponding to one nibble of the original sixteen nibble *state* matrix). These sixteen 4-bit lookup tables will reduce the memory requirement from 16,384 to 2048 bytes. However, now `subCells`, `siftRows` and `mixColumns` operation will be performed by 15 `xor`, 16 `and`, 15 `shift` and 16 `lookup` in each round.

3.7 Decryption

The block cipher mode of operations like Counter(CTR) and Output Feedback(OFB) support decryption of an encrypted message without actually implementing the inverse of the underlying block cipher. However, the use of such mode of operations may not always be possible. The block cipher may have to be incorporated in an existing cryptosystem which uses Cipher Block Chaining (CBC) mode of operation. With CBC, implementing the decryption routine of the block cipher becomes mandatory. The decryption of a block cipher is similar to encryption except that the inverse of each component is applied in reverse order. All the implementation techniques defined in Section 3 for encryption of LED are also applicable to its decryption routine. However, while implementing the decryption, a major question arises: "How to combine the inverse of substitution and permutation layer in a single lookup table?". In the forward direction(encryption), we computed the lookup tables by combining the effect of subCells and mixColumn operations of one round, but this is not possible for the reverse direction (decryption). So two possible alternatives are

- Decryption – 1: Implement the inverse of subCells and mixColumn operations in separate lookup tables. This almost doubles the number of lookup operations required to implement each round as compared to encryption routine and results in lower throughput.
- Decryption – 2: Combine the inverse subCells of roundi with inverse mixColumn operation of roundi-1. This way both the operations can be performed in single lookup as was done in encryption. However, now the values of interleaved operations such as the inverse of addConstants and addKey need to be recomputed and then added to the state. Moreover, mixColumn operation from the last round and subCells operation from the first round are still to be computed by separate non-combined lookup tables. So the two sets of lookup tables are to be stored. This almost doubles the memory requirement but results in higher throughput as compared to the Decryption – 1 method.

4 Results & Discussion

We implemented LED block cipher for all six implementation types (Section 3), in four programming languages (Java, C#, C++, Python). Separate dedicated implementations were coded for three key settings (64, 80 and 128 bit) by pre-computing lookup tables for each. Figure 4 summarizes the implementation details. All 72 implementations were then run on Elastic Compute (EC2) instances of AWS cloud with operating system Windows and Linux for both 64-bit and 32-bit versions (E). Detail of programming languages and IDEs are given in Table 4. Mono (open-source implementation of Microsoft's .NET Framework) was used to run C# code on Linux. Subsequently, we benchmarked the implementations for throughput (KB/s) and Software Efficiency (SE) [36] on four different platforms for the scenario discussed in section 1. For a fair comparison, we used similar coding conventions across all programming languages. SE was calculated as

$$SE = \frac{Throughput[KB/s]}{CodeSize[KB]} \quad where\ KB = 1024\ bytes.$$

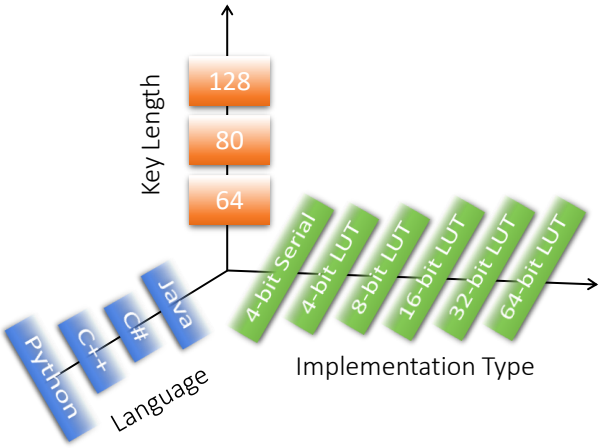

Figure 4: Implementation Breakdown.

Table 4: Programming Languages and integrated development environment used for code implementation.

#	IDE	Language	Version
1	Net Beans 8.2	Java	Java 1.8
2	Visual Studio 2017	C#	.Net 3.5
3	Code::Blocks 17.2	C++	GCC 5.1
4	PyCharm Community 2018.2	Python	Python 3.7

4.1 Operations and Memory

Figure 5 shows the memory and number of basic operations required for implementation types explained in Section 3.

- The figure shows that *larger the number of precomputed tables (memory requirement), lesser the number of operations required to perform the encryption.* However, 16-bit LUT implementation is an exception. It requires both memory and number of operations lesser than the 8-bit LUT implementation. This is because of the huge difference in the size of lookup tables with 4-bit and 8-bit input (16 vs 256 byte).

- *There is also a pack and unpack cost associated with each implementation.* The plaintext and key bytes need to be packed according to format and data type required by each implementation. Subsequently, the ciphertext has to be unpacked back to bytes at the end of encryption. This packing-unpacking involves `and`, `shift` and `xor` operations. Consequently, overall throughput has been measured after taking into account the packing-unpacking cost.

- If *state* is stored in data types larger than 8-bits, then more number of `shift` & `and` operations are required to select the appropriate chunk of the *state* for lookup. The number of these operations can also increase as a result of applying permutation to the *state* e.g. `shiftRows` in 8-bit LUT implementation.

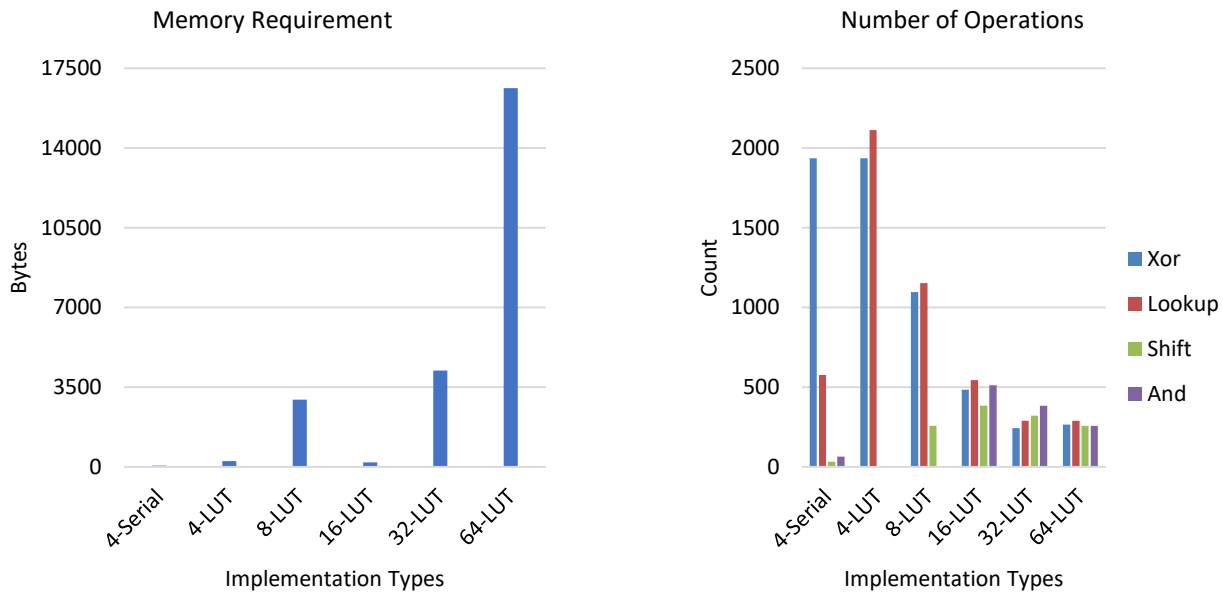

Figure 5: Memory and Number of basic operations required for LED-64 Encryption foreach implementation type.

4.2 Throughput

Figure 6 shows the throughput of all six implementations of LED-64 on 32 & 64-bit versions of Windows and Linux operating system.

- The 64-bit LUT implementation in C++ on 64-bit Linux outperforms all other implementations. At the other end of the throughput spectrum, 4-bit Serial implementation in Python on 32-bit Linux is the slowest one.
- The 64-bit LUT implementation achieves maximum throughput in almost all programming languages and platforms except in Java for the 32-bit version of Windows and Linux where 32-bit LUT implementation performs slightly better.
- The Java implementations for all types and platforms are faster than the same implementations in the remaining three languages with an exception of 64-bit LUT implementation in C++. None of the Python implementations achieve the throughput of 100KB/s thus making these the slowest. Python is an interpreted higher level language which does not convert the code logic to native code at compile time, rather code is interpreted to native code at runtime. However, standard distributions of other interpreted languages (Java & C#) include a Just In Time(JIT) compiler which converts the bytecode to native code at runtime, thus making these faster from Python.
- Maximum and minimum throughput of each implementation in all programming languages is achieved while running the code on 64-bit Linux and 32-bit Linux respectively. The C# implementations perform reasonably well on both 64-bit Linux despite the fact that these were run through mono since the .Net Framework is native to Windows. However, throughput of same implementations is much lower on 32-bit Linux as compared to 32-bit Windows.

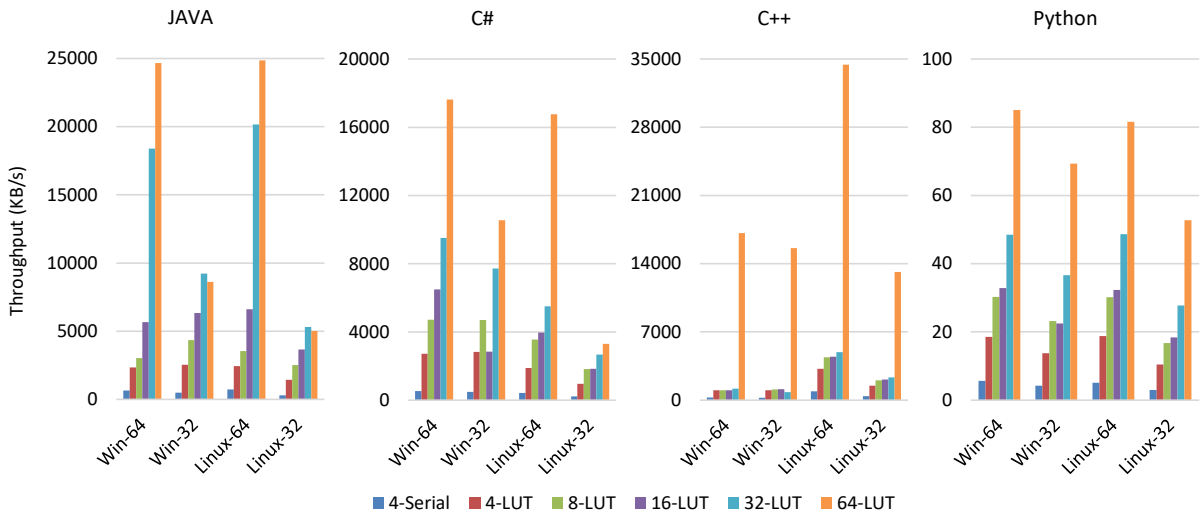

Figure 6: Throughput of all Six Implementations of LED-64 in four programming languages on different Operating Systems

4.3 SE Benchmark

Figure 7 shows SE of all six implementations of LED-64 on Windows and Linux for both 32 & 64-bit versions.

- Although the 64-bit LUT implementation attains the highest throughput, it is not the most efficient one, because it requires large memory for precomputed lookup tables.
- The implementation with highest SE on almost all platforms and programming languages is 16-bit LUT implementation with an exception of C++ implementations on Windows for both 64 and 32-bit version. This is because of low throughput of C++ implementations in Windows.
- The Java implementations for all types and operating systems have higher SE when compared to similar implementations in other languages except 8-bit LUT. This is because of the reason that Java has no 8-bit data type to store an unsigned 8-bit value. So higher 16-bit data type is used to store lookup tables of 8-bit LUT implementation. This almost doubles the memory required for lookup tables and results in lower SE. However Java SE 8 and later do have support for unsigned integer(32-bit) and long(64-bit) data types so there is no additional memory requirement for lookup tables of 32-bit LUT and 64-bit LUT implementations as compared to other programming languages.
- Similar to throughput, the Python implementations have lowest SE as compared to other languages.
- For 64-bit LUT and 32-bit LUT implementations, highest SE is achieved on Linux-64 by C++ and Java implementations respectively.

Results show the amount of impact the choice of programming language and platform has on implementation of LED block cipher. Implementation of the same block cipher algorithm produces remarkably different throughput based on these choices. This is

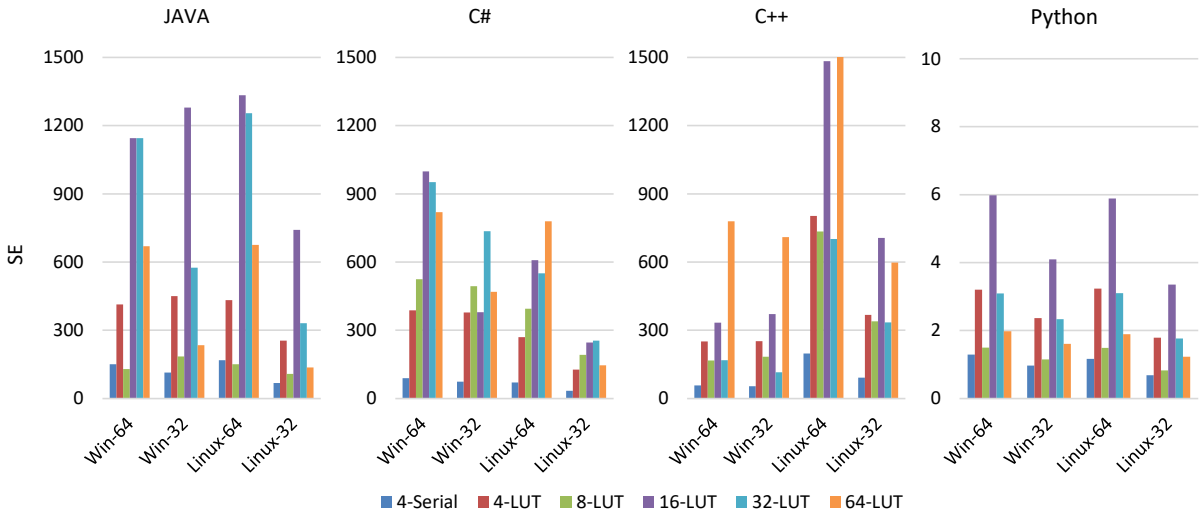

Figure 7: Software Efficiency (SE) of all Six Implementations of LED-64 in four programming languages on different Operating Systems.

because of the fact that different programming languages process the same piece of logic differently on different platforms which can have significant impacts upon execution time. Moreover, the same implementation entails different memory requirements in different programming languages based on availability or lack of support for particular native data type. These implementations have also highlighted that packing-unpacking cost is an additional consideration factor towards the overall throughput particularly in scenarios where small chunks of data needs to be processed in a key-agile environment. F shows the throughput and SE for 80 and 128 bit key lengths.

The main goal of the study was to highlight the impact of choices relating to implementation type, programming language, and platform. That is why we explained all implementation techniques for only one block cipher. However the implementation techniques mentioned in Section 3 can be easily extended to implement other lightweight block ciphers and impact of these choices will be similar for other SPN/ Feistel lightweight block ciphers. As a proof of concept, we implemented PRESENT-80 [18] (ISO standard for lightweight block ciphers [29]) for 4-bit LUT based implementation in all four programming languages (Java, C#, C++, Python). Figure 8 shows the throughput of these implementations on 64-bit Windows and Linux. The impact of these choices on PRESENT is quite similar to that on LED block cipher. The C# implementation performed better than Java on Windows and Java implementation performed better than C# on Linux. The Python implementations were the slowest and C++ implementation in Linux performed better than it did in Windows.

In order to see the impact of the two choices for implementing decryption routine, we implemented 4-bit LUT decryption for LED-64 in all four programming languages. G shows the comparison between Throughput and Software Efficiency of encryption and both the decryption methods for 4-bit LUT implementation on all four platforms. Since method-1 requires more number of lookup operations to implement one round of decryption, its throughput is slower than that of encryption and method-2. On the other hand, the throughput of decryption method-2 is comparable to encryption routine and the marginal difference between these is because of the implementation overhead for recomputing the key values in the decryption routine. However, the SE for method-2 is much lesser than the encryption and decryption method-1 because of the fact that method-2

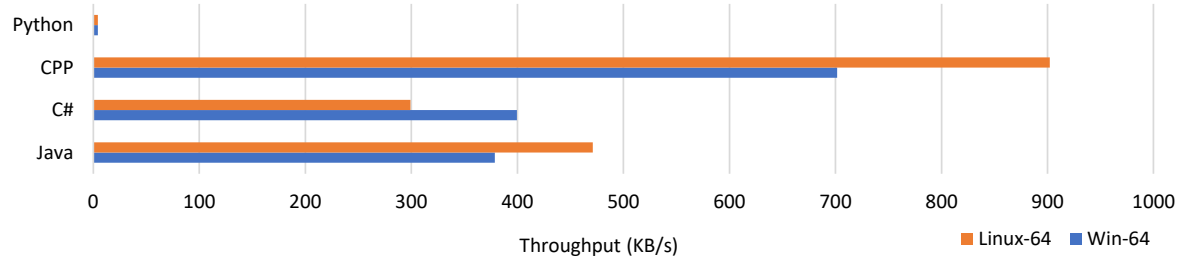

Figure 8: Throughput of 4-bit LUT based Implementation of PRESENT-80 in four programming languages on 64-bit Windows and Linux.

requires more memory resources. For 4-bit LUT implementation, the memory cost increases by the factor of two as two sets of the lookup tables are to be stored. However, for other implementations like 64-bit LUT, this factor can be reduced by using larger tables for combined lookup of subCells & mixColumns operations and smaller tables for non-combined lookups. Similarly a vast range of larger combined and smaller non-combined lookup table combinations is possible for implementing decryption routine.

5 Related & Future Work

Both, hardware and software implementations of lightweight cryptogtraphy (symmetric & asymmetric) were evaluated in [28, 47]. The authors provided a classification of lightweight schemes for estimating their suitability in different types of embedded systems. In [55], authors discussed lightweight hardware and software cryptographic solutions for Wireless Sensor Networks. The focus was towards *how the hardware can influence* the usage of lightweight cryptographic primitives in commercial and research based wireless sensor nodes and how the software implementations handle it. A comparative study of hardware implementations of block ciphers was presented in [43] which included area, power consumption, and throughput as the performance metric. In [27], the authors reported on the performance of 12 lightweight block ciphers in ATtiny45 (a low power, high-performance 8-bit microcontroller from AVR series of ATMEL). The block ciphers were implemented in assembly language for both encryption and decryption routines and compared for code size, RAM and cycles. A framework to benchmark lightweight block ciphers for RAM footprint, execution time and binary code size was provided in [24]. In [20], authors benchmarked software implementations of 12 lightweight and 5 conventional block ciphers on a mixed signal microcontroller (MSP430) for CPU cycles, energy consumption, and memory requirement. A comparison of 20 different software oriented block ciphers on smart phones for two forms of implementation (native JAVA APIs and Sponge Castle API) was given in [46]. The authors reported on current consumption and execution time of these implementations for different message lengths. The two standards for block ciphers AES [21] and PRESENT [18] were evaluated in the context of security applications on smart phones in [3]. A comprehensive survey of 52 block ciphers and their 360 lightweight implementations for both hardware and software was provided in [36]. Software implementations of three lightweight block ciphers PRESENT, LED and PICOLO were compared for performance on `x86` architecture in [12]. In [57], authors presented implementation of Low-power Encryption Algorithm (LEA)[37] in javascript language. Unlike other programming languages, javascript does not have support for

unsigned integer and rotation operations so authors discussed different techniques for solving the issue. The implementation was benchmarked for cycles per byte in different Web browsers on devices ranging from personal computers to mobile devices.

5.1 Future Work

In this paper, implementation techniques are explained for LED block cipher which introduced many novel ideas relating to block cipher design. LED does not employ any key schedule and user supplied key is used as-is where required. However, It has large number of rounds to thwart against the related key attack which makes it slower as compared to other block ciphers [36]. On the other hand, many lightweight block ciphers specify a proper key schedule which may or may not use the components from block cipher encryption routine [18, 6, 9, 52]. These key-schedule algorithms often use bit permutations which are very easy to realize in hardware as compare to the software. Comparing different lightweight block ciphers with a lesser number of rounds and proper key schedule may provide good insight about how a key schedule may affect the usability of a block cipher in key-agile environments.

The two main disadvantages of precomputed lookup table based implementations are large memory requirements and the possibility of cache side-channel attacks(CSCA), because lookup from the table depends on the portion of the master key. The former is easy to solve as developers can choose between larger or smaller lookup tables depending upon the environment in which cryptosystem is being deployed. However, thwarting against CSCA involves using the masking techniques which may have a performance hit. The cache is a small memory where frequently accessed portions of data are stored so that data can be served faster to the CPU on subsequent requests. Cache hit or miss occurs if a particular entry in the cache is found or not [48]. This cache hit and miss phenomena enables an attacker to perform trace-based cache side-channel attacks to recover the key [1]. Cache-Timing attacks recover the key by analyzing execution time for known-plaintexts [13]. One possible way forward is to analyze the implementation techniques explained in Section 3 for CSCA and improve these to resist CSCA without much compromise on performance. Another line of work is to test these implementations on platforms with smaller caches which may produce different results. Because the EC2 instances used in this paper have sufficiently large cache which enabled the implementations with larger memory requirement like 64-bit LUT to reside in cache memory completely.

The current trends in lightweight cryptography include designing ciphers which support decryption with minimal overhead [2], have low latency [19] or provide authenticated encryption [26]. The portfolio 1 of CESEAR competition deals with authenticated encryption for lightweight applications and recently NIST has also published a list of 56 lightweight block ciphers as round – 1 candidate for its lightweight cryptography project. Extending the implementation techniques from this paper to these lightweight block ciphers and authenticated encryption schemes will quantify their usability across different choices of platform and programming languages.

6 Conclusion

The significance of efficient and robust encryption in resource constrained devices is driving considerable research as well as development. This is attributed to the proliferation/utility of IoT devices on one hand and security/privacy concerns on the other. The 64-bit

block length and support for smaller key sizes make lightweight block ciphers ideal for small resource constrained devices while catering for software implementation suitability requirements for high-end devices and servers in network systems. Inherently, designers of lightweight cryptography have to balance trade-offs between security, cost, and performance; amongst which it is generally easier to optimize any two at a time. This paper has further highlighted the impact *choice of the platform and programming language* has on the performance of a cryptographic primitive. We discussed six different lookup-table based software implementation techniques for lightweight block ciphers along with their relevant trade-offs. We implemented LED block cipher for three key settings *(64, 80, 128 bits)* in four programming languages *(Java, C#, C++, Python)* for all six implementation types. We benchmarked these 72 implementations for throughput and software efficiency on four different platforms, thereby providing a quantitative elaboration of the impact accrued from the choice of programming language, platform and implementation type. These implementations are envisaged to serve as a crypto library which can be referenced for future research and analysis but the use of these codes in production without taking measures against SCA is not recommended.

References

[1] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on aes (short paper). In *International Conference on Information and Communications Security*, pages 112–121. Springer, 2006.

[2] Martin R Albrecht, Benedikt Driessen, Elif Bilge Kavun, Gregor Leander, Christof Paar, and Tolga Yalçın. Block ciphers–focus on the linear layer (feat. pride). In *International Cryptology Conference*, pages 57–76. Springer, 2014.

[3] Carlos Andrés, Morales-Sandoval Miguel, Díaz-Pérez Arturo, et al. An evaluation of aes and present ciphers for lightweight cryptography on smartphones. In *Electronics, Communications and Computers (CONIELECOMP), 2016 International Conference on*, pages 87–93. IEEE, 2016.

[4] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A lightweight hash. In *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 1–15. Springer Berlin Heidelberg, 2010.

[5] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: a block cipher for low energy. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 411–436. Springer, 2014.

[6] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. Gift: a small present. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 321–345. Springer, 2017.

[7] Zhenzhen Bao, Peng Luo, and Dongdai Lin. Bitsliced implementations of the prince, led and rectangle block ciphers on avr 8-bit microcontrollers. In *International Conference on Information and Communications Security*, pages 18–36. Springer, 2015.

[8] Lejla Batina, Amitabh Das, Barış Ege, Elif Bilge Kavun, Nele Mentens, Christof Paar, Ingrid Verbauwhede, and Tolga Yalçın. Dietary recommendations for lightweight block ciphers: power, energy and area analysis of recently developed architectures. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 103–112. Springer, 2013.

[9] Adnan Baysal and Sühap Şahin. Roadrunner: A small and fast bitslice block cipher for low cost 8-bit processors. In *International Workshop on Lightweight Cryptography for Security and Privacy*, pages 58–76. Springer, 2015.

[10] Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis Wingers. The simon and speck lightweight block ciphers. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.

[11] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In *Advances in Cryptology – CRYPTO 2016*, pages 123–153. Springer Berlin Heidelberg, 2016.

[12] Ryad Benadjila, Jian Guo, Victor Lomné, and Thomas Peyrin. Implementing lightweight block ciphers on x86 architectures. In *International Conference on Selected Areas in Cryptography*, pages 324–351. Springer, 2013.

[13] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[14] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3(30), 2009.

[15] Eli Biham. A fast new des implementation in software. In *International Workshop on Fast Software Encryption*, pages 260–272. Springer, 1997.

[16] Alex Biryukov and David Wagner. Slide attacks. In *International Workshop on Fast Software Encryption*, pages 245–259. Springer, 1999.

[17] Alex Biryukov and David Wagner. Advanced slide attacks. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 589–606. Springer, 2000.

[18] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. Present: An ultra-lightweight block cipher. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 450–466. Springer, 2007.

[19] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, et al. Prince–a low-latency block cipher for pervasive computing applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 208–225. Springer, 2012.

[20] Mickaël Cazorla, Kevin Marquet, and Marine Minier. Survey and benchmark of lightweight block ciphers for wireless sensor networks. In *Security and Cryptography (SECRYPT), 2013 International Conference on*, pages 1–6. IEEE, 2013.

[21] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.

[22] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[23] Christophe De Canniere. Trivium: A stream cipher construction inspired by block cipher design principles. In *International Conference on Information Security*, pages 171–186. Springer, 2006.

[24] Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Léo Perrin, Johann Großschädl, and Alex Biryukov. Triathlon of lightweight block ciphers for the internet of things. *Journal of Cryptographic Engineering*, pages 1–20, 2015.

[25] Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. Design strategies for arx with provable bounds: Sparx and lax. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 484–513. Springer, 2016.

[26] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1. 2. *Submission to the CAESAR Competition*, 2016.

[27] Thomas Eisenbarth, Zheng Gong, Tim Güneysu, Stefan Heyse, Sebastiaan Indesteege, Stéphanie Kerckhof, François Koeune, Tomislav Nad, Thomas Plos, Francesco Regazzoni, et al. Compact implementation and performance evaluation of block ciphers in attiny devices. In *International Conference on Cryptology in Africa*, pages 172–187. Springer, 2012.

[28] Thomas Eisenbarth, Sandeep Kumar, Christof Paar, Axel Poschmann, and Leif Uhsadel. A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers*, 24(6):522–533, nov 2007.

[29] International Organization for Standardization. Information technology - security techniques - lightweight cryptography - part 2: Block ciphers. *ISO/IEC 29192-2:2012*, 2012.

[30] Benoît Gérard, Vincent Grosso, María Naya-Plasencia, and François-Xavier Standaert. Block ciphers that are easier to mask: How far can we go? In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 383–399. Springer, 2013.

[31] Zheng Gong, Svetla Nikova, and Yee Wei Law. Klein: a new family of lightweight block ciphers. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 1–18. Springer, 2011.

[32] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varıcı. Ls-designs: Bitslice encryption for efficient masked software implementations. In *International Workshop on Fast Software Encryption*, pages 18–37. Springer, 2014.

[33] Jian Guo, Thomas Peyrin, and Axel Poschmann. The photon family of lightweight hash functions. In *Annual Cryptology Conference*, pages 222–239. Springer, 2011.

[34] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The LED block cipher. In *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 326–341. Springer Berlin Heidelberg, 2011.

[35] Mike Hamburg. Accelerating aes with vector permute instructions. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 18–32. Springer, 2009.

[36] George Hatzivasilis, Konstantinos Fysarakis, Ioannis Papaefstathiou, and Charalampos Manifavas. A review of lightweight block ciphers. *Journal of Cryptographic Engineering*, 8(2):141–184, 2018.

[37] Deukjo Hong, Jung-Keun Lee, Dong-Chan Kim, Daesung Kwon, Kwon Ho Ryu, and Dong-Geon Lee. Lea: A 128-bit block cipher for fast encryption on common processors. In *International Workshop on Information Security Applications*, pages 3–27. Springer, 2013.

[38] Deukjo Hong, Jaechul Sung, Seokhie Hong, Jongin Lim, Sangjin Lee, Bon-Seok Koo, Changhoon Lee, Donghoon Chang, Jesang Lee, Kitae Jeong, et al. Hight: A new block cipher suitable for low-resource device. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 46–59. Springer, 2006.

[39] Mark Hung. Leading the iot, gartner insights on how to lead in a connected world. *Gartner Research*, pages 1–29, 2017.

[40] Maryam Izadi, Babak Sadeghiyan, Seyed Saeed Sadeghian, and Hossein Arabnezhad Khanooki. Mibs: a new lightweight block cipher. In *International Conference on Cryptology and Network Security*, pages 334–348. Springer, 2009.

[41] Anthony Journault, François-Xavier Standaert, and Kerem Varici. Improving the security and efficiency of block ciphers based on ls-designs. *Designs, Codes and Cryptography*, 82(1-2):495–509, 2017.

[42] Ferhat Karakoç, Hüseyin Demirci, and A Emre Harmancı. Itubee: a software oriented lightweight block cipher. In *International Workshop on Lightweight Cryptography for Security and Privacy*, pages 16–27. Springer, 2013.

[43] Paris Kitsos, Nicolas Sklavos, Maria Parousi, and Athanassios N Skodras. A comparative study of hardware architectures for lightweight block ciphers. *Computers & Electrical Engineering*, 38(1):148–160, 2012.

[44] Manoj Kumar, Saibal K Pal, and Anupama Panigrahi. Few: A lightweight block cipher. *IACR Cryptology ePrint Archive*, 2014:326, 2014.

[45] Benjamin Lac, Anne Canteaut, Jacques JA Fournier, and Renaud Sirdey. Thwarting fault attacks against lightweight cryptography using simd instructions. In *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*, pages 1–5. IEEE, 2018.

[46] Lukas Malina, Vlastimil Clupek, Zdenek Martinasek, Jan Hajny, Kimio Oguchi, and Vaclav Zeman. Evaluation of software-oriented block ciphers on smartphones. In *Foundations and Practice of Security*, pages 353–368. Springer, 2014.

[47] Charalampos Manifavas, George Hatzivasilis, Konstantinos Fysarakis, and Konstantinos Rantos. Lightweight cryptography for embedded systems–a comparative analysis. In *Data Privacy Management and Autonomous Spontaneous Security*, pages 333–349. Springer, 2014.

[48] Heiko Mantel, Alexandra Weber, and Boris Köpf. A systematic study of cache side channels across aes implementations. In *International Symposium on Engineering Secure Software and Systems*, pages 213–230. Springer, 2017.

[49] Seiichi Matsuda and Shiho Moriai. Lightweight cryptography for the cloud: exploit the power of bitslice implementation. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 408–425. Springer, 2012.

[50] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: a very compact and a threshold implementation of aes. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 69–88. Springer, 2011.

[51] Taehwan Park, Hwajeong Seo, and Howon Kim. Fast implementation of simeck family block ciphers using avx2. In *2018 International Conference on Platform Technology and Service (PlatCon)*, pages 1–6. IEEE, 2018.

[52] Gilles Piret, Thomas Roche, and Claude Carlet. Picaro–a block cipher allowing efficient higher-order side-channel resistance. In *International Conference on Applied Cryptography and Network Security*, pages 311–328. Springer, 2012.

[53] Chester Rebeiro, David Selvakumar, and ASL Devi. Bitslice implementation of aes. In *International Conference on Cryptology and Network Security*, pages 203–212. Springer, 2006.

[54] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–427. Springer, 2010.

[55] Rodrigo Roman, Cristina Alcaraz, and Javier Lopez. A survey of cryptographic primitives and implementations for hardware-constrained sensor network nodes. *Mobile Networks and Applications*, 12(4):231–244, 2007.

[56] Klaus Schwab. *The fourth industrial revolution*. Crown Business, 2017.

[57] Hwajeong Seo and Howon Kim. Low-power encryption algorithm block cipher in javascript. *Journal of information and communication convergence engineering*, 12(4):252–256, 2014.

[58] Adi Shamir. Squash–a new mac with provable security properties for highly constrained devices such as rfid tags. In *International Workshop on Fast Software Encryption*, pages 144–157. Springer, 2008.

[59] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. Piccolo: an ultra-lightweight blockcipher. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 342–357. Springer, 2011.

[60] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-bit blockcipher clefia. In *International Workshop on Fast Software Encryption*, pages 181–195. Springer, 2007.

[61] Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka, and Eita Kobayashi. TWINE: A lightweight block cipher for multiple platforms. In *Selected Areas in Cryptography*, pages 339–354. Springer Berlin Heidelberg, 2013.

[62] Gangqiang Yang, Bo Zhu, Valentin Suder, Mark D Aagaard, and Guang Gong. The simeck family of lightweight block ciphers. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 307–329. Springer, 2015.

[63] Muhammad Reza Z'aba, Norziana Jamil, Mohd Ezanee Rusli, Md Zaini Jamaludin, and Ahmad Azlan Mohd Yasir. I-presenttm: An involutive lightweight block cipher. *Journal of Information Security*, 5(03):114, 2014.

[64] Wentao Zhang, Zhenzhen Bao, Dongdai Lin, Vincent Rijmen, Bohan Yang, and Ingrid Verbauwhede. Rectangle: a bit-slice lightweight block cipher suitable for multiple platforms. *Science China Information Sciences*, 58(12):1–15, 2015.

A LED Test Vectors

The test vectors of LED block cipher for three key lengths(64, 80 & 128 bits) are given below in hexadecimal byte notation.

64-bit key	
plaintext	10 32 54 76 98 BA DC FE
key	10 32 54 76 98 BA DC FE
ciphertext	0A 30 55 E1 83 39 CF 85

80-bit key	
plaintext	10 32 54 76 98 BA DC FE
key	10 32 54 76 98 BA DC FE 10 32
ciphertext	5D 2E 9A 88 90 C2 E1 B0

128-bit key	
plaintext	10 32 54 76 98 BA DC FE
key	10 32 54 76 98 BA DC FE 10 32 54 76 98 BA DC FE
ciphertext	6D 8B 42 85 F7 10 F4 2C

Table 5: Hardware implementation results of LED block cipher.

Key Length	Technology	Area[GE]	Throughput[Kbits/s]	Ref
64	0.18μm	966	5.1	[34]
80	0.18μm	1040	3.4	[34]
128	0.18μm	1265	3.4	[34]
128	0.18μm	3194	133	[8]

B Finite Field Multiplication

A nibble(4-bit) can be represented as polynomial with bits as coefficients in GF(2).

$$b_3b_2b_1b_0 \mapsto b(x)$$
$$b(x) = b_3x^3 + b_2x^2 + b_1x + b_0$$

According to this, the polynomial representation of 2 is x and field multiplication is performed as

$$\begin{aligned} b.x &= ((b_3x^3 + b_2x^2 + b_1x + b_0)\ .\ x)\ mod\ (x^4 + x + 1) \\ &= (b_3x^4 + b_2x^3 + b_1x^2 + b_0x)\ mod\ (x^4 + x + 1) \\ &= b_2x^3 + b_1x^2 + (b_0 \oplus b_3)x + b_3 \end{aligned}$$

The modulo operation is performed if degree of resultant polynomial is greater than 3 which is dependent upon the bit b_3. Thus multiplication with 2 is performed by a left shift of one bit and conditional xor with $x + 1$.

Table 6: Software implementation results of LED block cipher.

Key Length	Architecture	Clock Speed	Throughput[KB/s]	Ref
64	4 – bit micro-controller	500 KHz	0.078	[34]
128	4 – bit micro-controller	500 KHz	0.052	[34]
80	8 – bit micro-controller	4 MHz	0.225	[27]
80	16 – bit micro-controller	4 MHz	0.215	[27]
80	32 – bit micro-controller	4 MHz	0.766	[27]
128	Intel Core i7 CPU Q720	1.6 GHz	18168.6	[12]
128	Intel XEON X5650	2.67 GHz	24621.5	[12]
64	AWS EC2 t2.micro instance	2.8 GHz	34429.4	this paper
80	AWS EC2 t2.micro instance	2.8 GHz	21178.9	this paper
128	AWS EC2 t2.micro instance	2.8 GHz	23675.2	this paper

C 4-bit Serial Implementation Code Listings

```
void AddConstants(byte[] state, int round)  {
state[0] ^= keySizeConst0;
state[4] ^= keySizeConst1;
state[8] ^= keySizeConst2;
state[12] ^= keySizeConst3;

byte temp = (byte)(RC[round] >> 3 & 0x07);
state[1] ^= temp;
state[9] ^= temp;

temp = (byte)(RC[round] & 0x07);
state[5] ^= temp;
state[13] ^= temp;
}
```

Listing 8: Add Constants

```
void SubCellShiftRows(byte[] state)  {
byte[] temp = new byte[16];

temp[0] = sbox[state[0]];
temp[1] = sbox[state[1]];
temp[2] = sbox[state[2]];
temp[3] = sbox[state[3]];
temp[4] = sbox[state[5]];
temp[5] = sbox[state[6]];
.
.
.
temp[14] = sbox[state[13]];
temp[15] = sbox[state[14]];

state = temp;
}
```

Listing 9: Sub Cells & Shift Rows

```
void MultiplyColumn(byte[] t)   {
//t is one column of 4x4 state matrix
t[0] = (x2Times(t[0]) ^ t[1] ^ xTimes(t[2]) ^ xTimes(t[3]));
t[1] = (x2Times(t[1]) ^ t[2] ^ xTimes(t[3]) ^ xTimes(t[0]));
t[2] = (x2Times(t[2]) ^ t[3] ^ xTimes(t[0]) ^ xTimes(t[1]));
t[3] = (x2Times(t[3]) ^ t[0] ^ xTimes(t[1]) ^ xTimes(t[2]));
}
```

Listing 10: xTimes - Multiply one Column by matrix A

D Multiplication Table 64-bit LUT Implementation

index	T_0	T_1
0	00BB00DD00AA0055	BB00DD00AA005500
1	00BA00D100AE0057	BA00D100AE005700
2	00BC00DF00A5005B	BC00DF00A5005B00
3	00B500D900A7005A	B500D900A7005A00
⋮	⋮	⋮
⋮	⋮	⋮
254	0042005B00380084	42005B0038008400
255	0044005500330088	4400550033008800

E AWS EC2 instance Details

	EC2 Type	Operating System	vCPUs	Memory(GB)
1	t2.micro	Windows Server 2008 R2 Base SP1 64-bit	1	1
2	t2.micro	Windows Server 2008 SP2 32-bit	1	1
3	t2.micro	Ubuntu Server 14.04 LTS 64-bit	1	1
4	t1.micro	Ubuntu 14.04-i386-server 32-bit	1	0.613

F Throughput & SE for LED-80 and LED-128

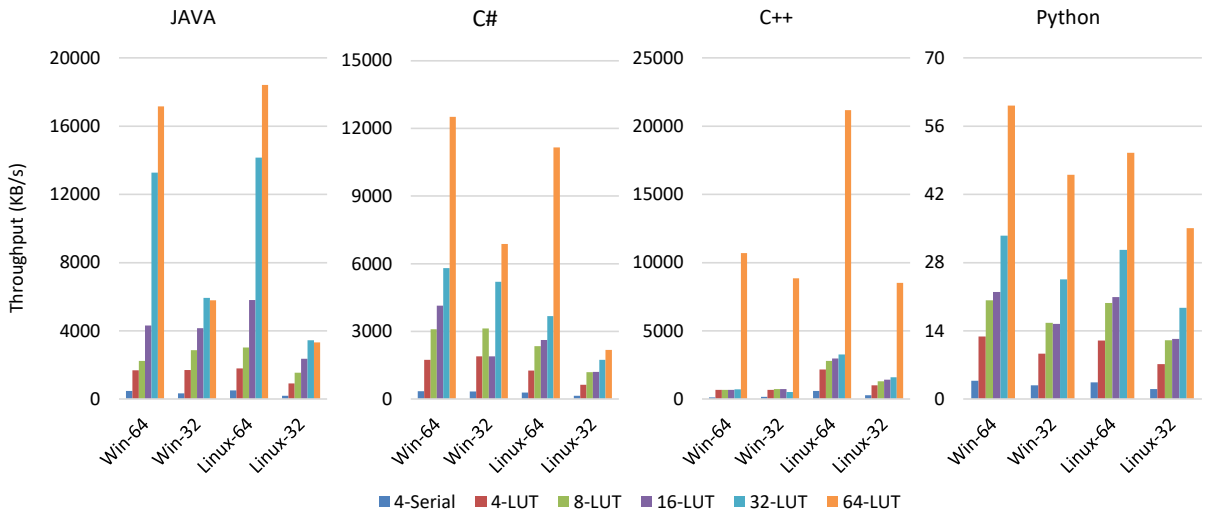

Figure 9: Throughput of all Six Implementations of LED-80 in four programming languages on different Operating Systems.

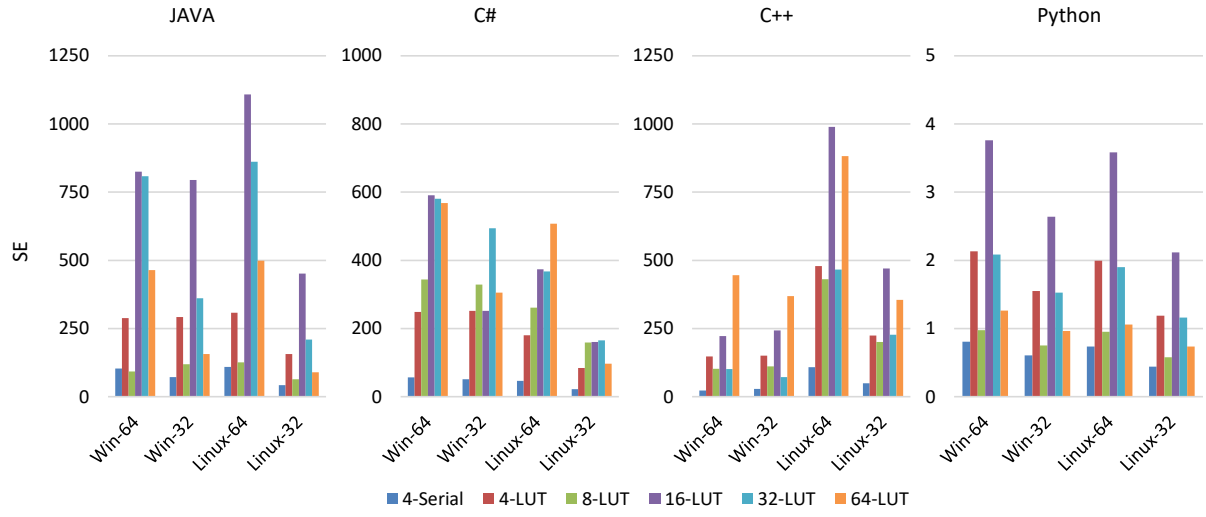

Figure 10: Software Efficiency of all Six Implementations of LED-80 in four programming languages on different Operating Systems.

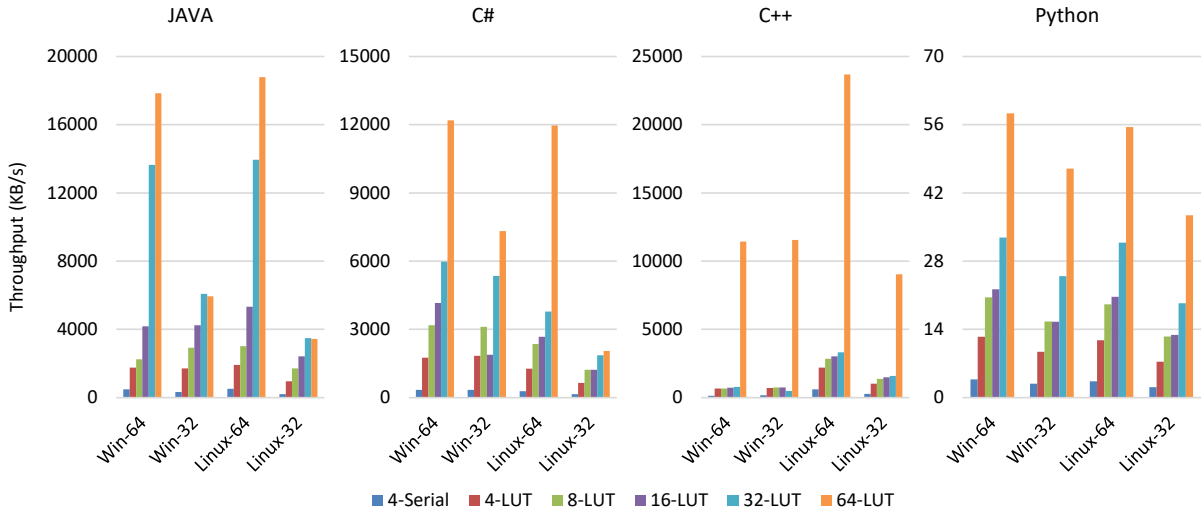

Figure 11: Throughput of all Six Implementations of LED-128 in four programming languages on different Operating Systems.

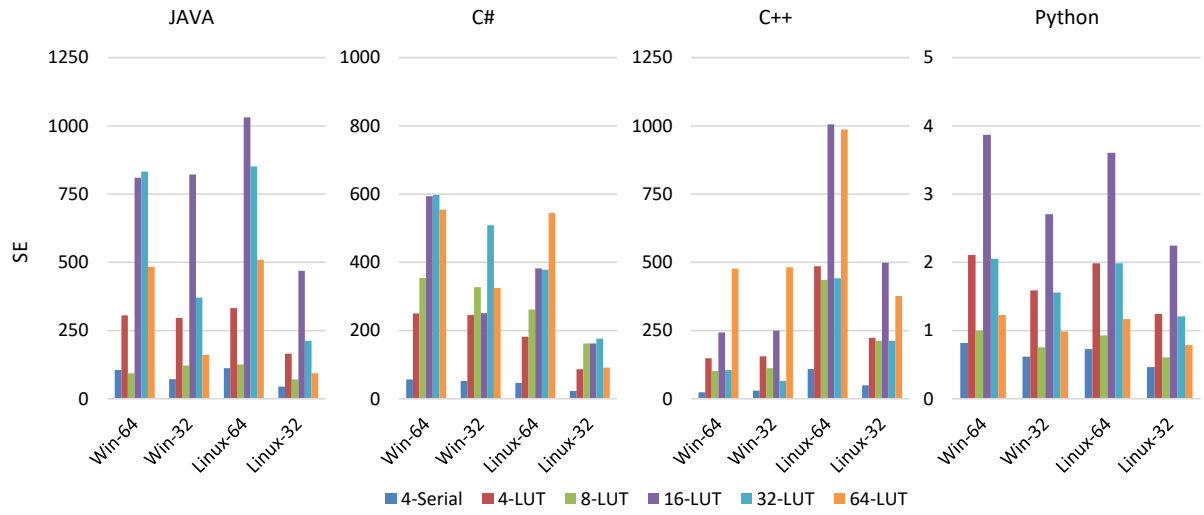

Figure 12: Software Efficiency of all Six Implementations of LED-128 in four programming languages on different Operating Systems.

G Throughput & SE for LED-64 Decryption

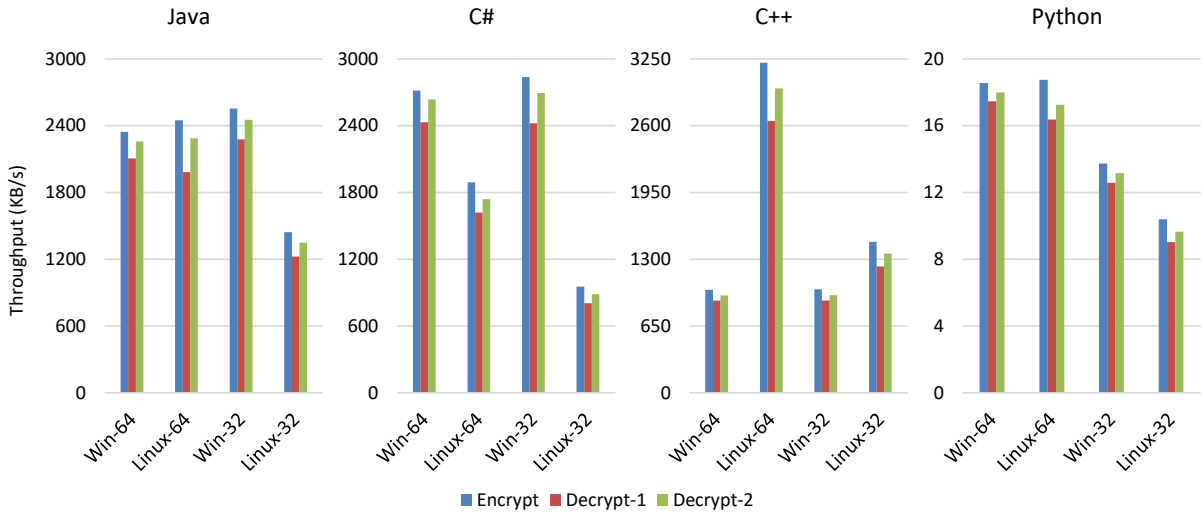

Figure 13: Comparison between Throughput of 4-bit LUT based Encryption and Decryption for LED-64 in all four programming languages on different Operating Systems.

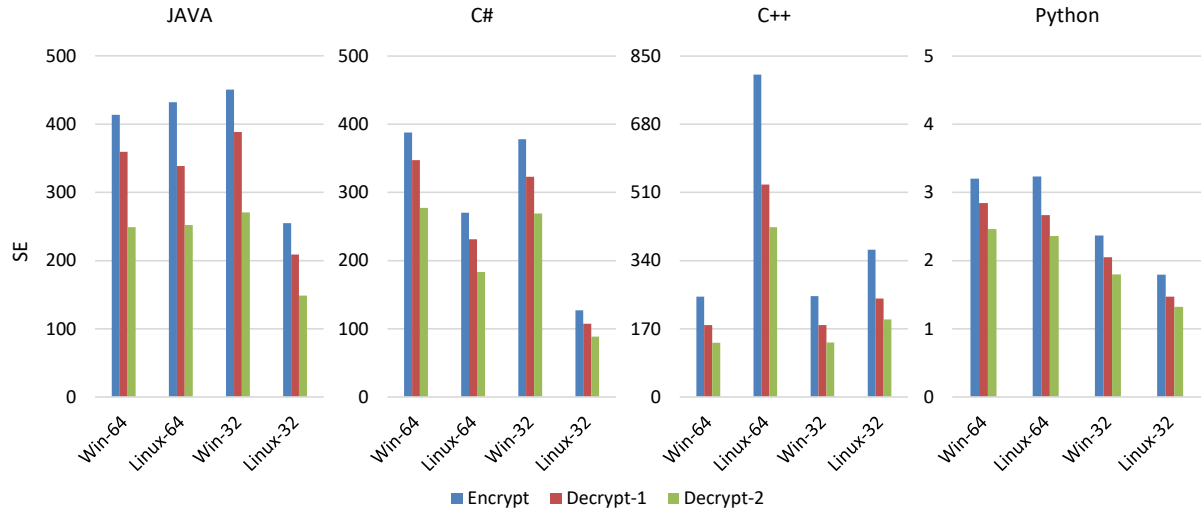

Figure 14: Comparison between Software Efficiency of 4-bit LUT based Encryption and Decryption for LED-64 in all four programming languages on different Operating Systems.