K-Cipher: A Low Latency, Bit Length Parameterizable Cipher

Michael Kounavis, Sergej Deutsch†, Santosh Ghosh†, and David Durham†

† Intel Labs, Intel Corporation, 2111, NE 25th Avenue, Hillsboro, OR 97124
Email: michael_kounavis@hotmail.com,
{sergej.deutsch, santosh.ghosh, david.durham}@intel.com

Rev. 0.1: November 2019, Rev. 0.5: February 2021

Abstract

We present the design of a novel low latency, bit length parameterizable cipher, called the "K-Cipher". K-Cipher is particularly useful to applications that need to support ultra low latency encryption at arbitrary ciphertext lengths. We can think of a range of networking, gaming and computing applications that may require encrypting data at unusual block lengths for many different reasons, such as to make space for other unencrypted state values. Furthermore, in modern applications, encryption is typically required to complete inside stringent time frames in order not to affect performance. K-Cipher has been designed to meet these requirements. In the paper we present the K-Cipher design and specification and discuss its security properties. Our analysis indicates that K-Cipher is secure against both known ciphertext, as well as adaptive chosen plaintext adversaries. Finally, we present synthesis results of 32-bit and 64-bit K-Cipher encrypt datapaths, produced using Intel's ® 10 nm process technology. Our results show that the encrypt datapaths can complete in no more than 767 psec, or 3 clocks in 3.9-4.9 GHz frequencies, and are associated with a maximum area requirement of 1875 μm^2.

1 Introduction

The paper presents a novel block cipher design which is lightweight, hardware efficient, as well as bit length parameterizable. Our cipher is called the "K-Cipher". In the K-Cipher design, the block length is not fixed (e.g., fixed to 128 bits), neither takes values from a small set of options (e.g., 64 or 128 bits). Instead the block length can take any arbitrary value between an upper and a lower bound and is an input parameter passed into the cipher.

The need for such block cipher comes from the requirements of modern applications. To address a wide range of vulnerabilities, applications employ cryptographic mechanisms to provide confidentiality and integrity. A common characteristic of applications is that their features are designed to work well with some existing or specially designed cipher (e.g., AES [2] or QARMA [6]).

In the paper we argue for the need of a new block cipher family that is more agile and easier to tune to the needs of a particular application or hardware. A new requirement we introduce is that the block length of a cipher should be an input parameter to the encryption operation. This should be the block length over which full confusion and diffusion operations are performed. Furthermore, the specification and security analysis of such block cipher should be, to some degree, block length independent. Such independence should substantially surpass what is accomplished today, where the block length is selected from a small number of options.

Our design supports encryption at arbitrary block lengths which, in our software prototype prototype take all values from 24 to 1024 at increments of 1.

The reason why we believe such requirement is important, is because applications operate on a variety of data of different lengths. The encrypted portions of such data may be of arbitrary lengths as well. Rather than designing an application with a cipher of fixed block length in mind, we argue for the opposite. That is, to have the ability to arbitrarily tune the block length of a block cipher to meet the needs of a particular application or hardware.

Another property of our cipher family is that it supports ultra low latency encryption in hardware. Specifically, our 32-bit and 64-bit encrypt datapaths can complete in 3 clocks in 3.9-4.9 GHz frequencies, where datapaths are synthesized using Intel's ® 10 nm process technology. It is fair to state that the landscape of today's block ciphers, which are either standard (e.g., AES [1]), or are to be standardized though the current NIST lightweight cryptography competition [7], does not include any cipher that simultaneously meets the two requirements stated: (i) support for full confusion and diffusion over arbitrary block lengths, as part of the encryption and decryption operations; and (ii) support for ultra low latency encryption and decryption operations in hardware.

For example, past lightweight cipher designs such as NSA's "Simon" and "Speck" [8], or designs like "PRINCE" [9] are not bit length parameterizable. Furthermore, such ciphers support critical paths, which can get even smaller with the design proposed here. For example, the PRINCE rounds, even though contain simpler SBox transformations, and simpler Mix Columns matrices when compared to AES, still require several clocks in the critical path, in typical client and server frequencies. Others ciphers like Simon employ a simple Feistel structure, which includes only elementary logical `AND`, `XOR` and rotation operations over 32-72 rounds. Furthermore, Simon supports only 5 fixed lengths: i.e., 32, 48, 64, 96 and 128 bits. Last, some submissions to the NIST lightweight cryptography competition, such as TinyJAMBU [11] or Xoodyak [12] support encryption at both very high speeds and arbitrary plaintext lengths. However, they do not fully diffuse the bits of their plaintext input into the bits of the ciphertext output. Instead, for a wide range of plaintext inputs, they merely add the input plaintext bits into the bits of some sponge state, in the finite field $\mathbb{F}_2$.

2 Overview

We envision that K-Cipher will be a useful tool for the encryption and decryption of data of varying lengths, performed by many different types of applications such as running in networking devices, gaming consoles, servers, low power clients and so on. K-Cipher supports fast encryption based on a novel confusion-diffusion network, which we discuss in this paper. The primitives it employs are: (i) block-wide addition with carries. In this operation, the carry-out bit is ignored. The operation is invertible, its inverse being subtraction with borrows. For the subtraction operation, K-Cipher just ignores the borrow-out bit; (ii) block-wide bit level reordering; and finally (iii) wide SBox substitution, which is realized as inversion in a binary Galois field. To further provide security against adaptive chosen plaintext adversaries, K-Cipher randomizes the SBox transformation, as explain in Section 4.2.

K-Cipher is designed to support confidentiality at desired security levels by employing the least possible number of rounds r, which, in the proposed design is set to 2 and 3. There are two encryption flows specified. First, a "Flex" flow is computationally lighter and suited to defend against known ciphertext adversaries. This one uses two rounds and deterministic SBox transformations. Second, a "CPA" flow is only incrementally more expensive, but defends against adaptive chosen plaintext adversaries. This one employs three rounds and randomized

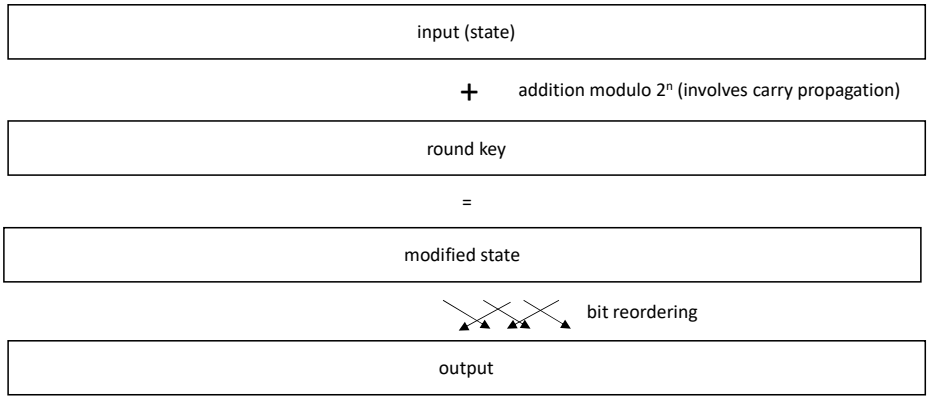

Figure 1: The Aggressive Adder Component of the K-Cipher Round

SBox transformations. The number of rounds can also be a configurable parameter. In the security analysis section of the paper, we discuss why the suggested numbers of rounds is a good choice for the cipher.

Substitution box lengths are determined by the block length. For example, to diffuse across 32 bits, K-Cipher uses 8-bit substitution boxes, as the 32-bit length is smaller than the square of the 8-bit length, and the minimum number of rounds is 2. Similarly, to diffuse across 128 bits, K-Cipher uses 16-bit substitution boxes, as the 128-bit length is smaller than the square of the 16-bit length. All primitives of the K-Cipher apply to a wide range of block lengths. When the number of rounds is three, the box widths guarantee that not only every bit value of the input is diffused over all bits of the output, but also every bit differential of the input can cause the appearance of bit differentials in every bit of the output. Arbitrary ciphertext lengths are supported using Galois field inverters of varying lengths, the sum of which is equal to the requested input and ciphertext lengths. In the K-Cipher specification discussed in Section 4, all employed Galois field inverters have fixed lengths but the last one, the length of which is determined by the block length, and a specific procedure configuring the cipher.

3 K-Cipher Design

3.1 Notation

In the design and specification sections that follow, we will be denoting by $[a_0 : a_1 : \ldots : a_{n-1}]$ a bit string of length n consisting of bits $a_0, \ldots, a_{n-1}$, where a_0 is the least significant bit of the string and a_{n-1} is its most significant bit. Similarly, we will be denoting by $[a_i : a_j]$ the substring of $[a_0 : a_1 : \ldots : a_{n-1}]$, which starts at bit position i and ends in bit position j. The bits at positions i and j are included in the substring. We will also be using the standard notation $[i, j]$ to refer to the set of integers from i to j. The operators '$\boxplus_M$'and '$\boxminus_M$'will denote addition and subtraction modulo M, whereas the operator $\oplus$ will denote XOR. Inversion in the finite field $\mathbb{F}_q$ will be denoted by $()_q^{-1}$. The assignment $\xleftarrow{\$}$ $\mathsf{Permutation}(e_0, e_1, \ldots, e_{v-1})$ will denote a random permutation of elements $e_0, e_1, \ldots, e_{v-1}$. Finally, the assignment notation $\xleftarrow{\$}$ $\mathsf{Range}(L, U)$ will denote assignment to a random uniformly distributed value in the set $[L, U]$.

3.2 The Aggressive Adder

Starting with Figure 1, we illustrate one of the basic components of the K-Cipher round called the "aggressive adder". The aggressive adder accepts as input some state, and then adds to this state a round key. The addition performed is not in the typical $\mathbb{F}_2$ arithmetic, but is in the integer arithmetic. Integer addition modulo 2^n, if seen as a bit-logical operation, performs strong mixing of its input bits, in order to produce the bits of the output. The mixing performed demonstrates regularity due to the use of carry values. By "mixing" in this document we mean computations on single bit values that involve a plurality of `AND`, `OR`, `NAND`, `NOR` or `XOR` operations.

For example let's consider that we add the numbers $[a_0 : a_1 : a_2; a_3]$ and $[b_0 : b_1 : b_2 : b_3]$ with each other and with some input carry value c_0. The first bit of the result is equal to $a_0 \oplus b_0 \oplus c_0$. The carry produced from the addition of the first two bits is equal to $a_0b_0 \oplus b_0c_0 \oplus a_0c_0$. Similarly, the second bit of the result is $a_1 \oplus b_1 \oplus a_0b_0 \oplus b_0c_0 \oplus a_0c_0$ and the carry produced from the addition of the second two bits is equal to $a_1b_1 \oplus a_1a_0b_0 \oplus a_1b_0c_0 \oplus a_1a_0c_0 \oplus b_1a_0b_0 \oplus b_1b_0c_0 \oplus b_1a_0c_0$. Moving on to the addition of the third least significant bits of the input, the same pattern of computation is repeated. The input bits are `XOR`-ed with each other and with the input carry, in order to produce the output bit. Furthermore, the input bits are multiplied with each other in $\mathbb{F}_2$ arithmetic (i.e., undergo a logical `AND` operation) and with the input carry and, subsequently, the products are `XOR`-ed with each other in order to produce the output carry. The third least significant bit of the result, as computed using this pattern, is $a_2 \oplus b_2 \oplus a_1b_1 \oplus a_1a_0b_0 \oplus a_1b_0c_0 \oplus a_1a_0c_0 \oplus b_1a_0b_0 \oplus b_1b_0c_0 \oplus b_1a_0c_0$. The third output carry is $a_2b_2 \oplus a_2a_1b_1 \oplus a_2a_1a_0b_0 \oplus a_2a_1b_0c_0 \oplus a_2a_1a_0c_0 \oplus a_2b_1a_0b_0 \oplus a_2b_1b_0c_0 \oplus a_2b_1a_0c_0 \oplus b_2a_1b_1 \oplus b_2a_1a_0b_0 \oplus b_2a_1b_0c_0 \oplus b_2a_1a_0c_0 \oplus b_2b_1a_0b_0 \oplus b_2b_1b_0c_0 \oplus b_2b_1a_0c_0$.

From the logical expressions above, it becomes evident that the mixing performed by the addition with carries stage, as measured by the number of $\mathbb{F}_2$ products which are `XOR`-ed with each other, gets only stronger as one moves from the least significant bit of the result toward the most significant bit. In fact, it grows stronger exponentially. It is easy to show that the n-th output bit of the result is produced by `XOR`-ing $2^n + 1$ terms, of which $2^n - 1$ are products.

To destroy the regularity which characterizes the addition with the carries stage, the aggressive adder performs a bit reordering operation on the addition output. Such reordering operation places the output bits coming from the integer adder in a seemingly random order, so that the number of $\mathbb{F}_2$ products of the logic equation of the result no longer increases monotonically but instead increases and decreases in a pseudorandom manner. Furthermore, the bit reordering operation aids the subsequent wide substitution stage, shown in Figures 2 and 3, ensuring that each bit of the output of the K-Cipher results from mixing all bits of the input with all bits of the key. The addition with carries is a bit length independent operation. Its specification is independent of the length of the inputs. It is also invertible, its inverse being the subtraction with borrows. Both of them are performed modulo 2^n. This means that any final carry-out or borrow-out signals produced from such operations are ignored.

3.3 Two Round K-Cipher Flow

Moving onto Figure 2, the processing steps of the Flex flow of the K-Cipher ar shown. The Flex flow employs two rounds, as shown in the figure. The Flex flow is introduced in this section, so that the functionality of the main K-Cipher components is explained. There is also the alternative CPA flow, which is similar, though it employs three rounds for better security. The differences are discussed in the next section. The first round consists of an aggressive adder stage that performs reordering using a first index sequence and a first round key denoted by

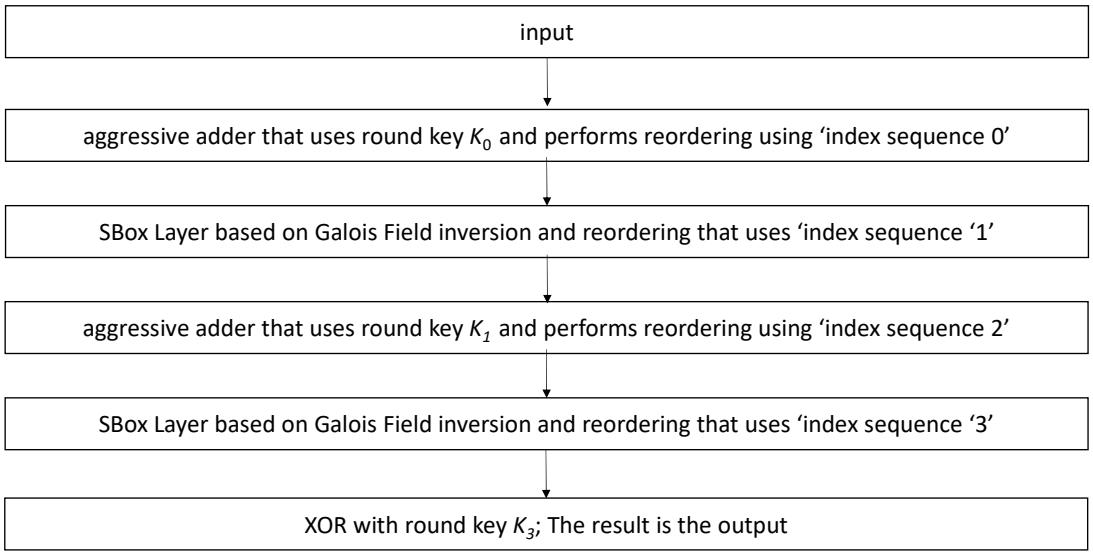

Figure 2: The Flex flow of the K-Cipher

K_0. The aggressive adder of the first round is followed by a wide substitution stage referred to as "SBox Layer" in the figure. The wide substitution stage is followed by yet another bit reordering stage. The second round consists of an aggressive adder stage also performing reordering using a different round key and index sequence. The round key is denoted by K_1. The second aggressive adder is followed by a second wide substitution stage and bit reordering. The processing steps of the K-Cipher conclude with an XOR operation performed between the cipher state and a third round key denoted by K_2.

The SBox layer of the Flex flow performs the following steps. It first divides its input n bits into blocks of m bits. Let's assume for now that n is a multiple of m. The cases where n is not a multiple of m are discussed further below. If n is a multiple of m, the SBox layer employs an array of $b \leftarrow n/m$ inverters in the $\mathbb{F}_{2^m}$ arithmetic, which replace their input bits with the bits of the inverse in $\mathbb{F}_{2^m}$. In the Flex flow, the output undergoes yet another bit reordering operation, like the one employed by the aggressive adder, but this reordering uses a different index sequence. Inversion in the Galois field arithmetic $\mathbb{F}_{2^m}$ is another operation supporting strong bit mixing. The mixing performed by the Galois field inverters employed by the K-Cipher does not demonstrate the regularity of addition with carries and is in fact pseudorandom. K-Cipher is designed to support strong encryption security by employing additions and inversions in two unrelated types of arithmetic (i.e., Galois field and integer) and by combining those into sequences of few rounds. Even though the additions and inversions may demonstrate imperfect differential distributions, their combination creates a much stronger cryptographic primitive. Moreover, the CPA flow mitigates adaptive chosen plaintext attacks by further randomizing the SBox transformation. So, not only the resulting ciphertext is produced from a difficult to solve non-linear system of equations, but also observed differential trails reveal little information about the state of the cipher. This aspect of our design is further discussed in the analysis section below.

The SBox layer, as defined so far, is bit length independent provided that the length of the state of the cipher n is a multiple of the width of the inverters employed m. In this case, the specification of the cipher is generic and each wide substitution stage employs n/m inverters. If n is not a multiple m, then these situations can be handled as shown in Figure 3. In the

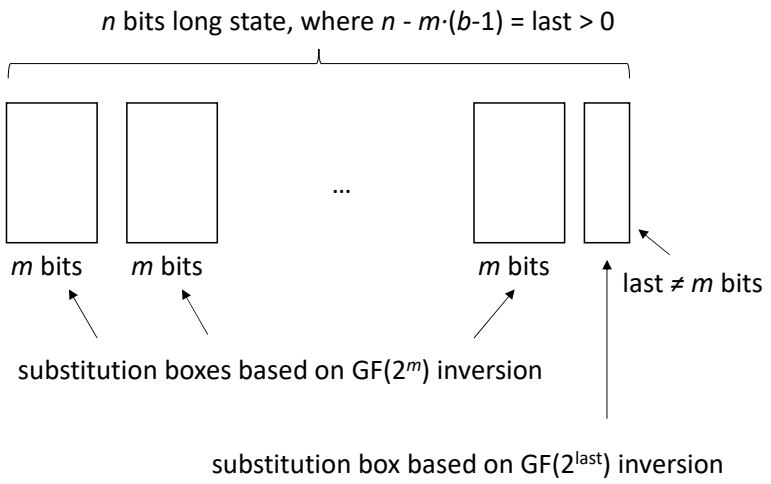

Figure 3: The SBox Layer of the K-Cipher

figure there are $b-1$ substitution boxes of width m which are employed, plus one more of width $\mathsf{last} \leftarrow n-(b-1)\cdot m$, where last is non-zero. The substitution stage employs $b-1$ inverters in the in $\mathbb{F}_{2^m}$ arithmetic and one inverter in the $\mathbb{F}_{2^{\mathsf{last}}}$ arithmetic handling the last bits of the cipher state.

The generation of the index sequences is done by an algorithm, which, in each iteration, determines the number of bits of an SBox that need to be distributed over all b substitution boxes. We refer to these bits of a substitution box as "bits-to-be-reordered". The number of times the algorithm needs to iterate for a particular box d is equal to $\lceil \frac{m}{b} \rceil$, if boxes are homogeneous. For each of the d iterations, the algorithm generates a random sequence of numbers. These are the indexes of the substitution boxes where the bits-to-be-reordered associated with the current iteration will be placed. Subsequently, for each bit-to-be-reordered the algorithm picks a bit position at random from among the empty bit positions in the input bit's target substitution box and assigns this position to the bit. This last step is repeated for all iterations of a triply nested loop executed by the algorithm.

One can show that the algorithm produces sequences of indexes that are both "correct" and "proper". By correct we mean that every bit of the input is placed in a different bit position of the output and there is no input which is omitted from the output. By proper we mean that if such reordering operations are combined with wide substitution operations, then, after $\log_m n$ rounds all bits of the input have been fully mixed with each other, even if additions with carries are absent.

3.4 Key Schedule and Use of Tweaks

For block lengths less than 32, the cipher's key is equal to the key schedule. For larger block lengths a key schedule algorithm is employed. The key schedule algorithm of the K-Cipher has the same structure as the cipher itself shown in Figure 2. However, it uses a different set of reordering sequences. Furthermore, instead of adding key bits into its state, it adds a plurality of constants. This is done in order for the cipher to expand the key into a sequence of round keys used by the cipher rounds.

Last, we discuss how the K-Cipher can be turned into a tweakable block cipher. This is done as follows: As said above, the key schedule of the K-Cipher involves 3 round keys. A tweak value is accepted which has the same length as each round key. The tweak value is added

to the first and the last round key of the K-Cipher key schedule using the aggressive adder algorithm of Figure 1. The result is a tweaked key schedule which is used by the encryption and decryption processes. The CPA flow uses, additionally, one more random bit string, which is part of the key and is called the "randomizer". The randomizer is not expanded into a key schedule.

4 K-Cipher Specification

4.1 Auxiliary Procedures

We begin the description of the cipher with auxiliary procedures that determine the widths of the substitution boxes for a particular block length value n. These are procedures DFunction(), EFunction(), GetLURange() and GetSBoxLengths() shown below. Procedure GetSBoxLengths() is the main one, which invokes the other three. Procedure DFunction() accepts as input an SBox width value m and determines whether this value is in a list of permitted SBox input lengths. These are the lengths of lines 1 and 2 of the pseudocode. Each length from the list is either a small prime, or can be written as a product of small primes. This property of the lengths allows for an efficient implementation of the Galois field inversion operation of the substitution boxes. Inversion in a characteristic two finite field $\mathbb{F}_{2^m}$, can be implemented in this case as inversion in an isomorphic composite field, where m is a permitted length. The irreducible polynomials defining the fields for each permitted length value are given in Appendix A.1. Suggested composite field representations for the characteristic two fields associated with each permitted length are given in Appendix A.2.

Procedure EFunction() determines whether an input box length m is greater or equal than a number of boxes b. Invoking this function is required in order to ensure that each bit string returned by a substitution box contains enough bits to be distributed to all substitution boxes of the cipher. This happens as part of a bit reordering operation. This is needed by both the Flex flow of K-Cipher, as well as the CPA flow. For the Flex flow, it guarantees that each bit of the input is diffused over all bits of the output. For the CPA flow it guarantees but each bit differential in the input can cause the appearance of bit differentials in every bit position of the output, provided that the number of rounds is at least 3.

```
DFunction(m)
1. PermittedLengths ← {12, 14, 15, 16, 18, 20, 21, 24, 25, 27, 28, 30, 32, 33, 34, 36, 38, 39}
2. PermittedLengths ← PermittedLengths ∪ [5, 10] ∪ {40, 42, 44, 45, 48, 49, 50, 52, 54, 55, 56, 60, 64}
3. if m ∈ PermittedLengths
4.     then
6.         return true
7.     else
8.         return false

EFunction(m, b)
1. if m ⩾ b
2.     then
3.         return true
4.     else
5.         return false

GetLURange(n)
1. if n ⩽ 256
```

```
2.      then
3.          return (8, 28)
4.      else
5.          return (16, 64)

GetSBoxLengths(n)
1. lengths_eq ← Ø
2. lengths_diff ← Ø
3. (L, U) ← GetLURange(n)
4. for l ← L to U do
5.      if DFunction(l) = false then continue
6.      b_l ← ⌈n/l⌉
7.      if EFunction(l, b_l) = false then continue
8.      score_l ← |l − b_l|
9.      if n − l · b_l = 0 then
10.         lengths_eq ← lengths_eq ∪ l
11.     else
12.         last_l ← n − l · (b_l − 1)
13.         if DFunction(last_l) = true and EFunction(last_l, b_l) = true then
14.             lengths_diff ← lengths_diff ∪ l
15.         else if DFunction(last_l + l) = true and EFunction(last_l + l, b_l − 1) = true then
16.             lengths_diff ← lengths_diff ∪ l
17.             score_l ← |l − b_l + 1|
18.             last_l ← last_l + l
19.             b_l ← b_l − 1
20. if lengths_eq ≠ Ø then
21.     m ← arg min_{l∈lengths_eq} score_l
22.     return (b_m, m, true, m)
23. else if lengths_diff ≠ Ø then
24.     m ← arg min_{l∈lengths_diff} score_l
25.     return (b_m, m, false, last_m)
26. else
27.     return ⊥
```

Procedure GetLURange() returns upper and lower bounds for SBox widths, associated with different block sizes. For block sizes up to 256 bits, box widths with may range between 8 and 28. For block sizes greater than 256 bits, box widths may range between 16 and 64.

Procedure GetSBoxLengths() searches for solutions to the problem of defining SBox widths. These solutions, on the one hand, should only include lengths from the permitted list and, on the other hand, should use as homogeneous boxes as possible. The types of solutions returned by this procedure can be two. In one type, all boxes have the same width. In another type all boxes have the same width, but the last which can be different. In lines 1 and 2 of the pseudocode, procedure GetSBoxLengths() initializes two lists of box lengths, one for each type of solution. In line 3, the procedure obtains upper and lower bounds L and U associated with box widths of block size n. A search for solutions begins in line 4. Solutions which do not satisfy the DFunction() are rejected in line 5. In line 6, a number of boxes b_l associated with a length value l is is computed as $b_l \leftarrow \lceil \frac{n}{l} \rceil$. Based on this number, a solution of length l is accepted, only if it satisfies procedure EFunction(). The check is done in line 7. In line 8, the procedure GetSBoxLengths() computes a score associated with the length value l. The score is defined as $\mathsf{score}_l \leftarrow |l - b_l|$. The score is defined in this way, so that solutions with too wide substitution boxes are penalized. The score rewards solutions that employ boxes which are as

wide as necessary, in order for procedures DFunction() and EFunction() to be satisfied.

In line 9, the procedure determines whether the block length n is divisible by the box width l. If this is the case, then the solution of length l is one where all boxes have the same length. In this case, length l is added to the list $\mathsf{lengths}_{eq}$. If this is not the case, the width of the last box must be different. Such width is computed as $\mathsf{last}_l \leftarrow n - l \cdot (b_l - 1)$ in line 12. In line 13, procedure GetSBoxLengths() checks whether the width of the last box last_l satisfies procedures DFunction() and EFunction(). If this is the case, the length value l is added to the list of solutions of the second type $\mathsf{lengths}_{diff}$. This is done in line 14. If the length value last_l does not satisfy one of DFunction() or EFunction(), the procedure checks if a last box width value, specified as the sum $\mathsf{last}_l + l$, happens to satisfy both of these procedures. This width value corresponds to a solution where there are only $b_l - 1$ boxes, among which the last box is wider, being of length $\mathsf{last}_l + l$ instead of last_l. In this case, the length l represents a valid solution of the second type. The number of boxes b_l, the last box length last_l and the score score_l are updated for this case in lines 17 through 19 of the code.

If the list $\mathsf{lengths}_{eq}$ is not empty, this means that a solution specifying boxes of identical widths exists. This solution is preferred over solutions of the second type. In this case, the solution m with the best score is determined in line 21 of the code, and returned in line 22. The code returns four values. These are the number of boxes b_m, the width m of all boxes but the last, a boolean indicating whether all boxes have the same width, and a last box width value last_m. If the list $\mathsf{lengths}_{eq}$ is empty, the procedure looks for a solution among the members of the list $\mathsf{lengths}_{diff}$. The solution with the best score is identified in line 24, and returned in line 25. If both lists are empty, the procedure returns the empty string in line 27. In fact, this never happened during the execution of our code, where the input block length n was between the values $n \leftarrow 24$ and $n \leftarrow 1024$. So, the code returned, either in line 22 or line 25.

Three additional auxiliary procedures are GetFlexKeyLength(), GetKeyLength() and GetRandomizer(). The first of the three, GetFlexKeyLength(), is directly invoked by the Flex flow of the K-Cipher, and indirectly by the CPA flow. It returns the length of the key bits used by the key expansion routine KeyExpansion() below, to produce a key schedule. There are five different key lengths, each associated with a different range of block lengths. These are lengths of 96, 128, 256, 512 and 1024 bits, respectively.

Procedure GetKeyLength() is invoked by both the Flex and the CPA flows. In line 1, it invokes GetFlexKeyLength(). For the Flex flow, the value l returned by this invocation is the length of the key of the cipher. If the flow f is equal to "Flex", then procedure GetKeyLength() returns in line 3. If the flow f is equal to "CPA", however, the key of the cipher includes both the key bits that produce a key schedule, as well as the bits of a randomizer. The length of the randomizer is 6 times the block length n. This is because there are 3 rounds in the CPA flow and each round uses two substrings of length n. Hence for the CPA flow, procedure GetKeyLength() returns the length value $l + 6n$.

Procedure GetRandomizer() accepts as input a block length value n and a key k. It obtains the bits of the randomizer string, which are those bits from k, located between bit positions l and $l + 6n - 1$. A bit string r, which is the randomizer, is returned in line 3.

GetFlexKeyLength(n)
1. **if** $n \leqslant 32$ **then return** 96
2. **else if** $n \leqslant 128$ **then return** 128
3. **else if** $n \leqslant 256$ **then return** 256
4. **else if** $n \leqslant 512$ **then return** 512
5. **else if** $n \leqslant 1024$ **then return** 1024

GetKeyLength(n, f)
1. $l \leftarrow$ GetFlexKeyLength(n)
2. **if** $f =$ Flex **then**
3. **return** l
4. **else**
5. **return** $l + 6n$

GetRandomizer(n, k)
1. $l \leftarrow$ GetFlexKeyLength(n)
2. $r \leftarrow k[l : l + 6n - 1]$
3. **return** r

4.2 K-Cipher Components

The SBox() procedure of the K-Cipher specification is shown below. In line 1, the procedure invokes procedure GetSBoxLengths() in order to obtain information about substitution box widths. If the flow of the cipher is CPA, then the procedure determines the starting positions of substrings of the randomizer, where random values aiding the substitutions to be performed are located. This is done in lines 3 and 4 of the code. The random values extracted from bit position u confuse each SBox's input. The random values extracted from bit position v randomize the SBox transformation itself. Specifically, the bits extracted from position v determine the choice of the SBox transformations which are used by the cipher. If the flow is Flex, then the SBox transformation is deterministic, being Galois field inversion. If the flow is CPA, however, each SBox is randomized, and the exact permutation to be applied is determined by the bits of the substring starting at position v.

The "for" loop of lines 5 to 21 performs the substitution steps for each box. The substitution associated with the last box, the width of which may be different, happens when the loop index i is equal to 0. The input to that box is the substring x_0 which is located between bit positions 0 and last $- 1$ of input x . If the flow of the cipher is Flex, then the SBox output is the inverse of x_0 in the finite field $\mathbb{F}_{2^{\text{last}}}$. This is computed in line 8. If, however, the flow of the cipher is CPA, then the procedure to be followed is more complex. First, two values r_0 and r_1 are extracted from the substrings of the randomizer located at positions u and v. This happens in lines 10 and 11 of the pseudocode. Then, a temporary variable t is computed from x_0 by XOR-ing x_0 with the substring r_0. Next, the result of the XOR operation is inverted in the finite field $\mathbb{F}_{2^{\text{last}}}$. The result of the inversion is added modulo 2^{last} to the randomizer value r_1. It is this last addition, which involves carry propagation, that randomizes the SBox transformation, making it difficult for an adversary to deduce the state of a substitution box by just observing a differential trail. The SBox output for x_0 is finally computed in line 13 by performing two shift operations on the temporary variable t. A similar procedure is followed for all subsequent SBox substitutions in lines 15 through 21 of the pseudocode. The computed string y is returned in line 22 of the pseudocode.

Procedure InvSBoxCPA() is applicable only to the CPA flow. Since the Flex flow is deterministic, and involves only Galois field inversions, the same procedure can be applied for both encrypting and decrypting data. In the CPA flow however, there is an asymmetry in the processing due to the introduction of addition with carries in the SBox. Hence, a different inverse SBox procedure must be followed. The flow of InvSBoxCPA() involves similar steps to those of procedure SBox(), though in reverse order. In line 1 box width information is obtained. In lines 2 and 3, the bit positions u and v are computed indicating the starting locations of the substrings of the randomizer containing r_0 and r_1. The main loop is between lines 4 and 16.

For $i = 0$, the inverse SBox input x_0 is located between bit positions 0 and $\mathsf{last} - 1$ of the input string x. Randomizer values r_0 and r_1 are obtained in a similar way in lines 7 and 8 for this box. Next, a temporary variable t is computed by first reversing the shift operations performed at the end of the SBox() transformation, and then subtracting modulo 2^{last} the randomizer value r_1. Next t is inverted in the finite field $\mathbb{F}_{2^{\mathsf{last}}}$ and the result of the inversion is XOR-ed with r_0. For all other boxes, a similar procedure is followed between lines 12 and 16 of the pseudocode. The computed string y is returned in line 17 of the pseudocode.

```
SBox(x, n, f, r, round)
1. (b, m, diff, last) ← GetSBoxLengths(n)
2. if f = CPA then
3.     u ← 2 · round · n
4.     v ← (2 · round + 1) · n
5. for i ← 0 to b − 1 do
6.     if i = 0 then
7.         x_0 ← x[0 : last − 1]
8.         if f = Flex then y[0 : last − 1] ← (x_0)^{-1}_{last}
9.         else
10.            r_0 ← r[u : u + last − 1]
11.            r_1 ← r[v : v + last − 1]
12.            t ← (x_0 ⊕ r_0)^{-1}_{last} ⊞_{last} r_1
13.            y[0 : last − 1] ← (t ≪_{last} 2) | (t ≫_{last} (last − 2))
14.    else
15.        x_i ← x[last + (i − 1) · m : last + i · m − 1]
16.        if f = Flex then y[last + (i − 1) · m : last + i · m − 1] ← (x_i)^{-1}_m
17.        else
18.            r_0 ← r[u + last + (i − 1) · m : u + last + i · m − 1]
19.            r_1 ← r[v + last + (i − 1) · m : v + last + i · m − 1]
20.            t ← (x_i ⊕ r_0)^{-1}_m ⊞_m r_1
21.            y[last + (i − 1) · m : last + i · m − 1] ← (t ≪_m 2) | (t ≫_m (m − 2))
22. return y
```

```
InvSBoxCPA(x, n, r, round)
1. (b, m, diff, last) ← GetSBoxLengths(n)
2. u ← 2 · round · n
3. v ← (2 · round + 1) · n
4. for i ← 0 to b − 1 do
5.     if i = 0 then
6.         x_0 ← x[0 : last − 1]
7.         r_0 ← r[u : u + last − 1]
8.         r_1 ← r[v : v + last − 1]
9.         t ← ((x_0 ≫_{last} 2) | (x_0 ≪_{last} (last − 2))) ⊟_{last} r_1
10.        y[0 : last − 1] ← (t)^{-1}_{last} ⊕ r_0
11.    else
12.        x_i ← x[last + (i − 1) · m : last + i · m − 1]
13.        r_0 ← r[u + last + (i − 1) · m : u + last + i · m − 1]
14.        r_1 ← r[v + last + (i − 1) · m : v + last + i · m − 1]
15.        t ← ((x_i ≫_m 2) | (x_i ≪_m (m − 2))) ⊟_m r_1
16.        y[last + (i − 1) · m : last + i · m − 1] ← (t)^{-1}_m ⊕ r_0
17. return y
```

Procedure BitReordering() performs the bit reordering steps of the K-Cipher specification. It accepts as input a string x, a block length value n, and an index order order specifying which

index sequence will be used for the reordering operation to be performed. For a particular block size n, the index order argument order may take integer values between 0 and 13. Orders 0, 1, 2 and 3 are used by the Flex and CPA encryption flows. Orders 4, 5, 6 and 7 are used by the KeyExpansion() procedure. Orders 8 and 9 are used by procedure TweakableKeyExpansion() which blends the bits of a tweak into the bits of a key schedule. Orders 10, 11, 12 and 13 are the inverse of orders 0, 1, 2 and 3 respectively, and are used by the decryption flows of K-Cipher. In line 1 of the procedure the input string x is represented as a sequence of bits $[x_0 : x_1 : \ldots : x_{n-1}]$. In line 2 of the pseudocode, the specified index sequence $\mathcal{R}_{n,\mathsf{order}}$ is represented as a set of integer values $\{R_0, R_1, \ldots, R_{n-1}\}$, denoting bit positions. The loop of lines 3 and 4 performs the actual bit reordering operation. The computed string y is returned in line 6.

Procedure KeyExpansion() computes the key schedule for both the Flex and CPA flows of the K-Cipher. If the block length n is less than, or equal to 32, then 96 bits of key k, passed as input, are sufficient to be used as bits of the key schedule. For block lengths up to 32, the procedure extracts $3n$ least significant bits from the key, and returns them as the bits of the key schedule of the cipher. For other block lengths, the procedure determines the length of the quantities over which expansion will take place. For block lengths between 33 and 64, the supplied key k is $L \leftarrow 128$ bits long. In this case, the expansion is performed over 64 bit quantities. The length of such quantities is denoted by w, and for these lengths w is equal to $L/2$. For block lengths greater than 64, the expansion is performed over quantities, the length w of which is equal to the key length L. The first round of the key expansion takes place in lines 6 to 9 of the pseudocode. In line 6, a constant $\mathcal{C}_1$ is added to w bits of key k modulo 2^w. In line 7, a bit reordering operation takes place using the index sequence 4, associated with the block length value w. Next, the substitution transformation SBox() of the Flex flow is invoked on the output of bit reordering. This transformation performs deterministic Galois field inversion. A second round of key expansion steps takes place, only for block lengths greater than 64, in lines 11 to 14 of the pseudocode. Three round keys K_0, K_1 and K_2 are finally returned in line 15 of the code.

The procedure TweakableKeyExpansion() accepts as input a block length value n, a key k and a tweak t. The tweak t is a bit string, the length of which is equal to the block length n. Procedure TweakableKeyExpansion() first invokes KeyExpansion() in line 1, obtaining three round keys K_0, K_1 and K_2. In lines 2 and 3, the bits of the tweak are blended into the bits of K_0. This is done by following steps similar to the steps of the key expansion procedure. Specifically, the key K_0 is first added to the tweak t modulo 2^n. The result of the addition undergoes a bit reordering operation, which uses the index sequence 8 and is specified for the block length n. The result of the bit reordering operation is a modified round key M_0. In lines 4 and 5, the same procedure is followed. However, the bit reordering operation uses the index sequence 9 this time. The bits of the tweak are blended into the bits of round key K_2, producing a modified round key M_2. The round keys M_0, K_1 and M_2 are returned in line 6.

BitReordering(x, n, order)
1. $[x_0 : x_1 : \ldots : x_{n-1}] \leftarrow x$
2. $\{R_0, R_1, \ldots, R_{n-1}\} \leftarrow \mathcal{R}_{n,\mathsf{order}}$
3. **for** $i \leftarrow 0$ **to** $n-1$ **do**
4. $\quad y_{R_i} \leftarrow x_i$
5. $y \leftarrow [y_0 : y_1 : \ldots : y_{n-1}]$
6. **return** y

KeyExpansion(n, k)
1. **if** $n \leqslant 32$ **then return** $(k[0:n-1], k[n:2n-1], k[2n:3n-1])$
2. $K \leftarrow k$
3. $L \leftarrow$ GetFlexKeyLength(n)
4. **if** $n \in [33, 64]$ **then** $w \leftarrow L/2$
5. **else** $w \leftarrow L$
6. $u \leftarrow K[0:w-1] \boxplus_w \mathcal{C}_1$
7. $v \leftarrow$ BitReordering$(u, w, 4)$
8. $u \leftarrow$ SBox$(v, w,$ Flex$, \perp, \perp)$
9. $K[L:L+w-1] \leftarrow$ BitReordering$(u, w, 5)$
10. **if** $n > 64$ **then**
11. $\quad u \leftarrow K[L:L+w-1] \boxplus_w \mathcal{C}_2$
12. $\quad v \leftarrow$ BitReordering$(u, w, 6)$
13. $\quad u \leftarrow$ SBox$(v, w,$ Flex$, \perp, \perp)$
14. $\quad K[L+w:L+2w-1] \leftarrow$ BitReordering$(u, w, 7)$
15. **return** $(K[0:n-1], K[n:2n-1], K[2n:3n-1])$

TweakableKeyExpansion(n, k, t)
1. $(K_0, K_1, K_2) \leftarrow$ KeyExpansion(n, k)
2. $u \leftarrow K_0 \boxplus_n t$
3. $M_0 \leftarrow$ BitReordering$(u, n, 8)$
4. $u \leftarrow K_2 \boxplus_n t$
5. $M_2 \leftarrow$ BitReordering$(u, n, 9)$
6. **return** $(M_0, K1, M_2)$

4.3 Encryption and Decryption Flows

The main encryption steps of the Flex flow are in procedure KCipherCoreEncFlex(). This procedure begins by adding the plaintext input x to a constant $\mathcal{C}_0$ modulo 2^n. Then, the output of the addition is added to the first round key K_0, modulo 2^n as well. In line 3 of the pseudocode, the output of the previous addition undergoes a first bit reordering operation, which uses the index sequence 0. Next, deterministic SBox() substitution takes place in line 4. This is based on Galois field inversion. A second bit reordering operation takes place in line 5 of the code, ending the first round of the Flex flow of the K-Cipher. The next round, shown in lines 6 to 9 is similar. Round key addition that uses the round key K_1 is performed in line 6. A third bit reordering operation is performed in line 7, followed by SBox() inline 8, and yet another bit reordering operation in line 9. The bit reordering operations of lines 7 and 9 use index sequences 2 and 3, respectively. The encryption flow ends with an XOR operation performed between the output of the second round and the round key K_2. The resulting bit string y is returned. The reverse steps of this sequence are included in procedure KCipherCoreDecFlex() which, on accepting as input round keys K_0, K_1 and K_2, decrypts y and returns x.

Procedure KCipherCoreEncCPA() implements the main encryption steps of the CPA flow. The main difference between the Flex and the CPA flows is that the CPA flow employs three rounds, instead of two, and SBox transformations are randomized, and not followed by reordering. In line 1 of the code, the plaintext is added to constant $\mathcal{C}_0$ modulo 2^n. In lines 2 to 4, the first round takes place that involves a round key addition step, a bit reordering step, and a randomized SBox step. Round key addition is performed modulo 2^n. The first round uses the round key K_0. The other two rounds use round keys K_1 and K_2 and take place in lines 5 through 7 and 8 through 10 of the pseudocode, respectively. A variable veil is computed in line 12 from round key K_2 by applying the bit reordering transformation and index sequence 3. This variable is XOR-ed with the output of the third round. The computed bit string y is

returned in line 13. These steps are reversed in procedure KCipherCoreDecCPA(), which on the same round key and randomizer inputs, decrypts y and returns x.

Finally, procedures KCipherEnc() and KCipherDec(), as well as their tweakable counterparts TweakableKCipherEnc() and TweakableKCipherDec(), are wrapper functions that compute round keys, extract randomizer bit strings, and, depending on the value of the flow parameter f, invoke either the processing steps of the Flex flow or the steps of the CPA flow.

KCipherCoreEncFlex(x, n, K_0, K_1, K_2)
1. $u \leftarrow x \boxplus_n \mathcal{C}_0$
2. $v \leftarrow u \boxplus_n K_0$
3. $u \leftarrow \mathsf{BitReordering}(v, n, 0)$
4. $v \leftarrow \mathsf{SBox}(u, n, \mathsf{Flex}, \perp, \perp)$
5. $u \leftarrow \mathsf{BitReordering}(v, n, 1)$
6. $v \leftarrow u \boxplus_n K_1$
7. $u \leftarrow \mathsf{BitReordering}(v, n, 2)$
8. $v \leftarrow \mathsf{SBox}(u, n, \mathsf{Flex}, \perp, \perp)$
9. $u \leftarrow \mathsf{BitReordering}(v, n, 3)$
10. $y \leftarrow u \oplus K_2$
11. **return** y

KCipherCoreDecFlex(y, n, K_0, K_1, K_2)
1. $u \leftarrow y \oplus K_2$
2. $v \leftarrow \mathsf{BitReordering}(u, n, 13)$
3. $u \leftarrow \mathsf{SBox}(v, n, \mathsf{Flex}, \perp, \perp)$
4. $v \leftarrow \mathsf{BitReordering}(u, n, 12)$
5. $u \leftarrow v \boxminus_n K_1$
6. $v \leftarrow \mathsf{BitReordering}(u, n, 11)$
7. $u \leftarrow \mathsf{SBox}(v, n, \mathsf{Flex}, \perp, \perp)$
8. $v \leftarrow \mathsf{BitReordering}(u, n, 10)$
9. $u \leftarrow v \boxminus_n K_0$
10. $x \leftarrow u \boxminus_n \mathcal{C}_0$
11. **return** x

KCipherCoreEncCPA(x, n, K_0, K_1, K_2, r)
1. $u \leftarrow x \boxplus_n \mathcal{C}_0$
2. $v \leftarrow u \boxplus_n K_0$
3. $u \leftarrow \mathsf{BitReordering}(v, n, 0)$
4. $v \leftarrow \mathsf{SBox}(u, n, \mathsf{CPA}, r, 0)$
5. $u \leftarrow v \boxplus_n K_1$
6. $v \leftarrow \mathsf{BitReordering}(u, n, 1)$
7. $u \leftarrow \mathsf{SBox}(v, n, \mathsf{CPA}, r, 1)$
8. $v \leftarrow u \boxplus_n K_2$
9. $u \leftarrow \mathsf{BitReordering}(v, n, 2)$
10. $v \leftarrow \mathsf{SBox}(u, n, \mathsf{CPA}, r, 2)$
11. $\mathsf{veil} \leftarrow \mathsf{BitReordering}(K_2, n, 3)$
12. $y \leftarrow v \oplus \mathsf{veil}$
13. **return** y

KCipherCoreDecCPA(y, n, K_0, K_1, K_2, r)
1. $\mathsf{veil} \leftarrow \mathsf{BitReordering}(K_2, n, 3)$
2. $u \leftarrow y \oplus \mathsf{veil}$
3. $v \leftarrow \mathsf{InvSBoxCPA}(u, n, r, 2)$
4. $u \leftarrow \mathsf{BitReordering}(v, n, 12)$
5. $v \leftarrow u \boxminus_n K_2$

6. $u \leftarrow \mathsf{InvSBoxCPA}(v, n, r, 1)$
7. $v \leftarrow \mathsf{BitReordering}(u, n, 11)$
8. $u \leftarrow v \boxminus_n K_1$
9. $v \leftarrow \mathsf{InvSBoxCPA}(u, n, r, 0)$
10. $u \leftarrow \mathsf{BitReordering}(v, n, 10)$
11. $v \leftarrow u \boxminus_n K_0$
12. $x \leftarrow v \boxminus_n \mathcal{C}_0$
13. **return** x

$\mathsf{KCipherEnc}(x, n, f, k)$
1. $(K_0, K_1, K_2) \leftarrow \mathsf{KeyExpansion}(n, k)$
2. **if** $f = \mathsf{Flex}$ **then**
3. $\quad y \leftarrow \mathsf{KCipherCoreEncFlex}(x, n, K_0, K_1, K_2)$
4. **else**
5. $\quad r \leftarrow \mathsf{GetRandomizer}(n, k)$
6. $\quad y \leftarrow \mathsf{KCipherCoreEncCPA}(x, n, K_0, K_1, K_2, r)$
7. **return** y

$\mathsf{KCipherDec}(y, n, f, k)$
1. $(K_0, K_1, K_2) \leftarrow \mathsf{KeyExpansion}(n, k)$
2. **if** $f = \mathsf{Flex}$ **then**
3. $\quad x \leftarrow \mathsf{KCipherCoreDecFlex}(y, n, K_0, K_1, K_2)$
4. **else**
5. $\quad r \leftarrow \mathsf{GetRandomizer}(n, k)$
6. $\quad x \leftarrow \mathsf{KCipherCoreDecCPA}(y, n, K_0, K_1, K_2, r)$
7. **return** x

$\mathsf{TweakableKCipherEnc}(x, n, f, k, t)$
1. $(K_0, K_1, K_2) \leftarrow \mathsf{TweakableKeyExpansion}(n, k, t)$
2. **if** $f = \mathsf{Flex}$ **then**
3. $\quad y \leftarrow \mathsf{KCipherCoreEncFlex}(x, n, K_0, K_1, K_2)$
4. **else**
5. $\quad r \leftarrow \mathsf{GetRandomizer}(n, k)$
6. $\quad y \leftarrow \mathsf{KCipherCoreEncCPA}(x, n, K_0, K_1, K_2, r)$
7. **return** y

$\mathsf{TweakableKCipherDec}(y, n, f, k, t)$
1. $(K_0, K_1, K_2) \leftarrow \mathsf{TweakableKeyExpansion}(n, k, t)$
2. **if** $f = \mathsf{Flex}$ **then**
3. $\quad x \leftarrow \mathsf{KCipherCoreDecFlex}(y, n, K_0, K_1, K_2)$
4. **else**
5. $\quad r \leftarrow \mathsf{GetRandomizer}(n, k)$
6. $\quad x \leftarrow \mathsf{KCipherCoreDecCPA}(y, n, K_0, K_1, K_2, r)$
7. **return** x

4.4 Generating Pseudorandom Index Sequences

Procedure BitReorderingIndexes(), shown below, computes pseudorandom index sequences. These sequences are used for bit reordering, and are associated with a specific block length value n. First, the returned indexes R_i, $i \in [0, n-1]$ are initialized to -1 in line 2 of the pseudocode. The source indexes, which have not yet been taken into account, are initialized in lines 3 and 4 for each of the boxes. Index placement happens in the "while" loop of lines 6 through 23. Inside this "while" loop, a second "for" loop places indexes from each of the boxes

of the cipher state in other boxes in a pseudorandom manner. The number of indexes to be placed in each iteration is computed in lines 10 and 11. For this computation the procedure emptySlotsInBoxes() is invoked, which determines the number of boxes that have available slots for index placement, as well as the indexes of these boxes. The per-box placement is done in lines 14 to 21, once the order of the available boxes is first altered in a pseudorandom manner in line 13. The loop stops when no new index has been placed in any of the iterations.

emptySlotsInBoxes$(n, R_0, R_1, \ldots, R_{n-1})$
1. $(b, m, \mathsf{diff}, \mathsf{last}) \leftarrow \mathsf{GetSBoxLengths}(n)$
2. $\mathcal{V} \leftarrow \emptyset$
3. $\mathcal{S} \leftarrow \emptyset$
4. $\mathsf{count} \leftarrow 0$
5. **for** $i \leftarrow 0$ **to** $b-1$ **do**
6. **if** $i = 0$ **then** $S_0 \leftarrow \sum_{j \in [0, \mathsf{last}-1], R_j = -1} 1$
7. **else** $S_i \leftarrow \sum_{j \in [\mathsf{last}+(i-1)\cdot m, \mathsf{last}+i\cdot m-1], R_j = -1} 1$
8. **if** $S_i > 0$ **then**
9. $\mathcal{V} \leftarrow \mathcal{V} \cup \{i\}$
10. $\mathcal{S} \leftarrow \mathcal{S} \cup \{S_i\}$
11. $\mathsf{count} \leftarrow \mathsf{count} + 1$
12. **return** $(\mathsf{count}, \mathcal{V}, \mathcal{S})$

GetPlace$(d, s, n, R_0, \ldots, R_{n-1})$
1. $(b, m, \mathsf{diff}, \mathsf{last}) \leftarrow \mathsf{GetSBoxLengths}(n)$
2. **if** $d = 0$ **then** $\mathsf{range} \leftarrow \mathsf{last}$
3. **else** $\mathsf{range} \leftarrow m$
4. $j \leftarrow 0$
5. **for** $i \leftarrow d$ **to** $d + \mathsf{range} - 1$ **do**
6. **if** $R_i = -1$ **then** $j \leftarrow j + 1$
7. **if** $s = j - 1$ **then return** $i - d$

BitReorderingIndexes(n)
1. $(b, m, \mathsf{diff}, \mathsf{last}) \leftarrow \mathsf{GetSBoxLengths}(n)$
2. **for** $i \leftarrow 0$ **to** $n-1$ **do** $R_i \leftarrow -1$
3. **for** $i \leftarrow 0$ **to** $b-1$ **do**
4. **if** $i = 0$ **then** $N_0 \leftarrow \mathsf{last}$ **else** $N_i \leftarrow m$
5. $\mathsf{allPlaced} \leftarrow \mathsf{false}$
6. **while** $\mathsf{allPlaced} = \mathsf{false}$ **do**
7. $\mathsf{atLeastOne} \leftarrow \mathsf{false}$
8. **for** $i \leftarrow 0$ **to** $b-1$ **do**
9. **if** $N_i = 0$ **then continue**
10. $(v, \{V_0, \ldots, V_{v-1}\}, \{S_0, \ldots, S_{v-1}\}) \leftarrow \mathsf{emptySlotsInBoxes}(R_0, \ldots, R_{n-1})$
11. $w \leftarrow \mathsf{min}(v, N_i, b)$
12. **if** $w > 0$ **then** $\mathsf{atLeastOne} \leftarrow \mathsf{true}$
13. $\{t_0, \ldots t_{v-1}\} \xleftarrow{\$} \mathsf{Permutation}(0, 1, \ldots, v-1)$
14. **for** $j \leftarrow 0$ **to** $w-1$ **do**
15. **if** $i = 0$ **then** $\mathsf{srcIndex} \leftarrow N_0 - w + j$
16. **else** $\mathsf{srcIndex} \leftarrow \mathsf{last} + (i-1)\cdot m + N_i - w + j$
17. **if** $V_{t_j} = 0$ **then** $\mathsf{dstIndex} \leftarrow 0$
18. **else** $\mathsf{dstIndex} \leftarrow \mathsf{last} + (V_{t_j} - 1)\cdot m$
19. $\mathsf{slotIndex} \xleftarrow{\$} \mathsf{Range}(0, S_{t_j} - 1)$
20. $\mathsf{placeInBox} \leftarrow \mathsf{GetPlace}(\mathsf{dstIndex}, \mathsf{slotIndex}, n, R_0, \ldots, R_{n-1})$
21. $R_{\mathsf{dstIndex}+\mathsf{placeInBox}} \leftarrow \mathsf{srcIndex}$
22. $N_i \leftarrow N_i - w$

23. **if** atLeastOne = false **then** allPlaced ← true
24. **return** $\{R_0, R_1, \ldots, R_{n-1}\}$

4.5 Cipher Constants

K-Cipher uses 15 constants. A single constant is used by block lengths of the range $[24, 32]$. For these block lengths, no key expansion takes place. Two constants are used by block lengths of the range $[33, 64]$. For these block lengths, encryption and decryption uses a first constant and key expansion a second constant. Three constants are used by each block length of the ranges $[65, 128]$, $[129, 256]$, $[257, 512]$, and $[512, 1024]$. For a particular block length range, constants are denoted by $\mathcal{C}_0$, $\mathcal{C}_1$ and $\mathcal{C}_2$ in the pseudocode above. The values of these constants are given in Appendix A.3.

5 Security Discussion

5.1 Security of the Flex Flow

The Flex flow of the K-Cipher is designed to address known ciphertext attacks. The adversary model for these attacks is as follows. The adversary can observe q ciphertexts $y^{(0)}, y^{(1)}, \ldots, y^{(q-1)}$. The adversary has no access to plaintext information or the key. The goal of the adversary is, after observing q ciphertexts, to determine what the encryption key is. We argue that the Flex flow is secure in this adversary model, because of the difficulty of solving the nonlinear system of equations relating the output of the Flex flow with the plaintext input and key.

Let's look at the properties of this nonlinear system. The algebraic degree of the system is equal to the sum of the number of plaintext bits n plus the key bits L. In fact, this algebraic degree value is reached quickly, as part of the K-Cipher processing. This is because of the carry propagation happening in each round. The resulting number of terms, which are XOR-ed with each other in the output equations of the Flex flow, is at least $\mathrm{O}(2^n)$, where n is the block size. Carry propagation over n bits results in terms which are produced by XOR-ing $2^n - 1$, $2^{n-1} - 1$, $2^{n-2} - 1$, etc., $\mathbb{F}_2$ products at bit positions $n - 1$, $n - 2$, etc., respectively. The last box contains the terms resulting from the strongest mixing operation. These terms are distributed over all boxes of the second round of the Flex flow and participate in Galois field inversion operations. Galois field inversions further mix these terms with the rest of the terms in a pseudorandom manner. We conjecture that, because of these reasons, the complexity of solving any linearized version of the resulting system of equations is at least the complexity of the brute force approach for breaking the cipher. By "brute force" complexity we mean any complexity of 2^n compute steps.

The Flex flow, however is vulnerable to known plaintext and known ciphertext attacks. The only mixing mechanism, which is block wide, is the carry propagation, which is statistical in nature. An attacker may exploit this vulnerability of the design and introduce differentials in a single SBox of the first round, which, on the second round can cause some boxes to be active and some to be inactive. The activations, or lack of, of boxes in the second round can be observed by the adversary. In this way, the adversary can obtain information about at least one differential trail associated with a first round SBox. Since, in the Flex flow, the SBox transformation is deterministic and based on Galois field inversion, knowledge about a differential trail can easily reveal the state of the SBox transformation. There can be at most 4 state values, with 2 on average, associated with a trail, provided that a trail is possible. To

address this vulnerability of the Flex flow, we extend the Flex flow in the form of the CPA flow. In what follows, we discuss why the CPA flow is secure against known plaintext and known ciphertext attacks, which can be adaptive as well.

5.2 Security of the CPA Flow

The adversary model we consider in this section is as follows. The adversary can adaptively submit q plaintexts $x^{(0)}, x^{(1)}, \ldots, x^{(q-1)}$ of his choice to the encryption oracle, and observe q ciphertexts $y^{(0)}, y^{(1)}, \ldots, y^{(q-1)}$. The adversary has no access to information about the key. The goal of the adversary is, once again, to determine what the encryption key is. This is an adaptive chosen plaintext attack. The analysis that follows is applicable to a similarly defined adaptive chosen ciphertext attack as well.

The security of the CPA flow stems from the fact that, once a differential trail is observed, there can be many different SBox permutations and state values, which may support this differential trail. This number is much larger than 4, for a single box, and, in fact, pushes the complexity of attacking the CPA flow back to the brute force complexity. Because of the fact that the number of rounds is 3, each single bit differential in the input can cause the appearance of bit differentials in every output bit of the cipher. On average, one should expect that half of the bits of the output will be perturbed. Each output box differential is characterized by some differential uncertainty. For a particular substitution box of width m, we define the output differential uncertainty $\mathcal{U}^{(m)}_{\mathsf{output}}$ as the minimum number of combinations of SBox permutations and state values, which may cause an output differential value to appear. For SBox transformations of widths 5, 6, 7 and 8, for instance, the output differential uncertainty $\mathcal{U}^{(m)}_{\mathsf{output}}$ is this equal to 400, 1664, 6628 and 26388, respectively.

Similarly, for each input box differential, associated with width m, we can define the input differential uncertainty $\mathcal{U}^{(m)}_{\mathsf{input}}$ as the minimum number of combinations of SBox permutations and state values, which are compatible with an input differential value. For example, for SBox width values of 5, 6, 7, and 8, the input differential uncertainty $\mathcal{U}^{(m)}_{\mathsf{input}}$ is equal to 246, 884, 3658, and 13898, respectively.

Since the number of rounds is 3, given a set of active boxes in the first and the third round, there is a number of combinations of permutations and state values in the middle round, which are compatible with an applied input differential and an observed output differential. We define the minimum trail uncertainty $\mathcal{U}^{(m)}_{\mathsf{trail}}$ of a single box of width m, as the minimum number of combinations of SBox permutations and state values for which a differential trail, associated with the box, is possible. For boxes of widths 5, 6, 7, and 8 the minimum trail and certainty $\mathcal{U}^{(m)}_{\mathsf{trail}}$ is equal to 8, 16, 16, and 32, respectively. In general, the minimum trail uncertainty is a conservative metric for the behavior of the randomized SBox transformations of the CPA flow. On average, the number of combinations of SBox transformations and state values, which can cause a differential trail to appear is much larger. For example, for 5, 6, 7, and 8-bit boxes, the average number permutations supporting a trail is 15, 29, 57, and 110, respectively. For our analysis, we also define the probability that a differential trail is possible. We have observed that, on boxes of width up to 8 bits, this probability is approximately 1/2. This property characterizes boxes of larger widths as well.

Based on what we discussed so far, we can compute the minimum input-output differential uncertainty $\mathcal{U}^{(b,m)}_{\mathsf{input\text{-}output}}$ of a CPA flow consisting of b homogeneous boxes of width m, and 3 rounds of the K-Cipher. This uncertainty is defined as the product between the input uncertainty $\mathcal{U}^{(m)}_{\mathsf{input}}$ and the output uncertainty $\mathcal{U}^{(m)}_{\mathsf{output}}$ raised to a power of $\frac{b}{2}$:

cipher	area (μm^2)	latency (psec)	number of clocks	freq.
K-Cipher Enc-32, Flex	614	**613**	3	4.9 GHz
K-Cipher Enc-64, Flex	1875	**767**	3	3.9 GHz

Figure 4: Synthesis results and performance of the K-Cipher Enc-32 and K-Cipher Enc-64 algorithms, when the encryption is performed by the Flex flow

$$\mathcal{U}^{(b,m)}_{\mathsf{input\text{-}output}} \leftarrow \mathcal{U}^{(m)}_{\mathsf{input}} \cdot (\mathcal{U}^{(m)}_{\mathsf{output}})^{\frac{b}{2}} \tag{5.1}$$

The exponent $\frac{b}{2}$ reflects the fact that, on average, there are $\frac{b}{2}$ boxes perturbed at the output of a differential trail, which originates from a single perturbed box.

From the minimum input-output differential uncertainty of a CPA flow, we can compute an approximation for the minimum number $\mathcal{N}^{(b,m)}_{\mathsf{input\text{-}output}}$ of combinations of SBox permutations and state values, which may be possible, given an observed input-output trail. To do this, we first multiply $\mathcal{U}^{(b,m)}_{\mathsf{input\text{-}output}}$ with the minimum trail uncertainty $\mathcal{U}^{(m)}_{\mathsf{trail}}$, raised to a power equal to the number of boxes divided by 2. This is done in order to take into account the fact that there is uncertainty, not only in the first and the third rounds but also in the middle as well. We conservatively estimate that only half of the middle round boxes will be active on average. Next, we multiply the result of the previous multiplication with the probability that all middle round differential trails are possible which is at least $(\frac{1}{2})^b$. The result is a reasonable approximation for the minimum number of combinations $\mathcal{N}^{(b,m)}_{\mathsf{input\text{-}output}}$ of SBox transformations and state values that can cause a differential trail to appear, in the case the CPA flow consists of homogeneous boxes:

$$\mathcal{N}^{(b,m)}_{\mathsf{input\text{-}output}} = \frac{\mathcal{U}^{(m)}_{\mathsf{input}} \cdot (\mathcal{U}^{(m)}_{\mathsf{output}})^{\frac{b}{2}} \cdot (\mathcal{U}^{(m)}_{\mathsf{trail}})^{\frac{b}{2}}}{2^b} \tag{5.2}$$

The security of the CPA flow stems from the fact that the number of combinations computed using this formula is above the brute force complexity for the range of K-Cipher block lengths.

5.3 Performance

We have built a software prototype of the Flex and CPA flows that supports encryption for all bit lengths from 24 to 1024 with increments of 1. Our optimized K-Cipher Enc-24 and K-Cipher Dec-24 Flex flow routines demonstrate a speed of 17 cycles per byte on an Intel ® Core ™ i7-8665U processor running at 1.9 GHz.

More interesting is the hardware performance of the K-Cipher Flex flow. K-Cipher employs standard integer adders, substitutions that can be implemented using lookup tables, and reordering operations that can be implemented using wires. These components add minimal latency to the critical path. Moreover, the number of these components is small for two and three rounds.

We have built and synthesized optimal encrypt and decrypt datapaths for the 32-bit and 64-bit versions of the K-Cipher Flex flow using Intel's ® 10 nm process technology. Our results are shown in the table of Figure 4. The area required by the encrypt and decrypt datapaths is 614 μm^2 and 1875 μm^2 respectively. Moreover, the complete encryption operation finishes in 617 psec and 767 psec respectively for the two datapaths, if there are no clocking constraints.

Finally, the implementations allow for the insertion of pipeline registers. The staged encryptions performed using these registers complete in 3 clocks. The minimum unconstrained clock periods supported correspond to frequency values of 4.9 and 3.9 GHz, respectively for the 32-bit and 64-bit datapaths.

6 Concluding Remarks

We presented the design and specification of a new cipher, called the K-Cipher. K-Cipher is bit length parameterizable and only involves few low latency components in the critical path, specifically integer adders and Galois field inverters. As such, the K-Cipher can be a useful tool for developing secure computer architecture components, where ultra low latency cryptographic mechanisms can be invoked inside the CPU. We have developed a software prototype of the cipher, which can successfully perform encryptions and decryptions for all block lengths from 24 bits up to 1024 bits, at block length increments of one, and hardware prototypes for some data paths.

As defined, the K-Cipher could also be used as part of other larger cryptographic constructions. Many different known cryptographic constructions could be employing the cipher, including Feistel structures, sponge structures, Davies Meyer constructions, modes such as CBC, CTR, XTS and so on. Finally, we note that this technical report extends a previous IEEE publication [13], which describes only the Flex flow of the cipher.

References

[1] *Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publication FIPS PUB 197.

[2] *Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices*, NIST Special Publication 800-38E.

[3] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue and U. Savagaonkar, *Innovative instructions and software model for isolated execution*, Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP), 2013.

[4] *AMD Secure Encrypted Virtualization (SEV)*, `https://developer.amd.com/sev/`, 2016.

[5] M. Rutland, *ARM v8.3 Pointer Authentication*, presentation, available online at `https://events.static.linuxfound.org/sites/events/files/slides/slides_23.pdf`, 2017.

[6] R. Avanzi, *The QARMA Block Cipher Family*, Cryptology ePrint Archive: Report 2016/444.

[7] *NIST Lightweight Cryptography Competition*, available online at `https://csrc.nist.gov/projects/lightweight-cryptography`

[8] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers *The Simon and Speck Families of Lightweight Block Ciphers*, Cryptology ePrint Archive: Report 2013/404.

[9] J. Borghoff, A. Canteaut, T. Guneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalcin, *PRINCE: a low-latency block cipher for pervasive computing applications*, ASIACRYPT 2012, Proceedings of the 18th international conference on The Theory and Application of Cryptology and Information Security, Pages 208-225, Beijing, China — December 02 - 06, 2012.

[10] Y. Dodis, T. Liu, M. Stam, J. Steinberger, *Indifferentiability of Confusion-Diffusion Networks*, hskip 1em plus 0.5em minus 0.4emCryptology ePrint Archive: Report 2015/680.

[11] H. Wu, and T. Huang, *TinyJAMBU: A Family of Lightweight Authenticated Encryption Algorithms*, Submission to the NIST Lightweight Cryptography Competition, available online at `https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/TinyJAMBU-spec.pdf`.

[12] J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, *Xoodyak, a lightweight cryptographic scheme*, Submission to the NIST Lightweight Cryptography Competition, available online at `https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/Xoodyak-spec.pdf`.

[13] M. Kounavis, S. Deutsch S. Ghosh, and D. Durham, *K-Cipher: A Low Latency, Bit Length Parameterizable Cipher*, Proceedings of 25th IEEE Symposium on Computers and Communications (ISCC), 2020.

Appendix

A.1 The polynomials defining the finite fields of SBox

The polynomials defining the finite fields of the SBox transformation are given in binary form below. The most significant bit of the polynomials is omitted.

```
uint32_t gf_inv_polys_range_5_32[28] =
{
    0x00000005, 0x00000003, 0x00000003, 0x0000001b, 0x00000003, 0x00000009, 0x00000005, 0x00000009,
    0x0000001b, 0x00000021, 0x00000003, 0x0000002b, 0x00000009, 0x00000009, 0x00000027, 0x00000009,
    0x00000005, 0x00000003, 0x00000021, 0x0000001b, 0x00000009, 0x0000001b, 0x00000027, 0x00000003,
    0x00000005, 0x00000003, 0x00000009, 0x0000008d
};

uint32_t gf_inv_polys_range_33_64[32][2] =
{
    {0x0000004b, 0x00000000}, {0x0000001b, 0x00000000}, {0x00000005, 0x00000000}, {0x00000035, 0x00000000},
    {0x0000003f, 0x00000000}, {0x00000063, 0x00000000}, {0x00000011, 0x00000000}, {0x00000039, 0x00000000},
    {0x00000009, 0x00000000}, {0x00000027, 0x00000000}, {0x00000059, 0x00000000}, {0x00000021, 0x00000000},
    {0x0000001b, 0x00000000}, {0x00000003, 0x00000000}, {0x00000021, 0x00000000}, {0x0000002d, 0x00000000},
    {0x00000071, 0x00000000}, {0x0000001d, 0x00000000}, {0x0000004b, 0x00000000}, {0x00000009, 0x00000000},
    {0x00000047, 0x00000000}, {0x0000007d, 0x00000000}, {0x00000047, 0x00000000}, {0x00000095, 0x00000000},
    {0x00000011, 0x00000000}, {0x00000063, 0x00000000}, {0x0000007b, 0x00000000}, {0x00000003, 0x00000000},
    {0x00000027, 0x00000000}, {0x00000069, 0x00000000}, {0x00000003, 0x00000000}, {0x0000001b, 0x00000000}
};
```

A.2 Suggested composite field realizations

Composite field isomorphisms for the finite fields of SBox follow from expressing each permitted box width as a product of primes or small numbers:

```
9 -> 3X3
10 -> 2X5
12 -> 3X6
14 -> 2X7
15 -> 3X5
16 -> 2X8
18 -> 3X6
20 -> 4X5
21 -> 3X7
24 -> 3X8
25 -> 5X5
27 -> 3X9
28 -> 4X7
30 -> 5x6
32 -> 4X8
33 -> 3X11
34 -> 2X17
36 -> 4x9
38 -> 2X19
39 -> 3X13
40 -> 5x8
42 -> 6X7
44 -> 5X11
45 -> 5x9
48 -> 6x8
49 -> 7X7
50 -> 2X5X5
52 -> 4X13
54 -> 6X9
55 -> 5X11
56 -> 7X8
60 -> 3X4X5
64 -> 8x8
```

A.3 K-Cipher Constants

K-Cipher uses following 15 constants. Byte sequences are represented using the little endian ordering. This means that the least significant byte of constant $\mathcal{C}_0$ of the block range $[65, 128]$ is `0x37`. The most significant byte of the same constant is `0xf9`.

```
uint32_t __k_cipher_range_24_32_const_0 = 0x820390b6;

uint64_t __k_cipher_range_33_64_const_0 = 0x010abcfe1d620c9a;
uint64_t __k_cipher_range_33_64_const_1 = 0xa99cac23bfb4f3ad;

uint64_t __k_cipher_range_65_128_const_0[2] = {
0x5f63c0ec346ddc37, 0xf98c63bbbbefa08e };
uint64_t __k_cipher_range_65_128_const_1[2] = {
0x44aa7cb19f6d53a0, 0x43daa42d7323101a };
uint64_t __k_cipher_range_65_128_const_2[2] = {
0xb3b27e401ae99fd0, 0x84177319f57a5e1b };

uint64_t __k_cipher_range_129_256_const_0[4] = {
0x49d69cc59cc874f8, 0x95f9f51b2856dafe, 0x98cdbb6d6554111d, 0x51702d3a34e78fdf };
uint64_t __k_cipher_range_129_256_const_1[4] = {
0x0cf17165dac7d437, 0xde2b49aa38844cc9, 0xd0ad712895e8d678, 0x97db930c1683f6fa };
uint64_t __k_cipher_range_129_256_const_2[4] = {
0xf61ee062c9489499, 0xc73550005b576a0e, 0x90b4719065e20f7d, 0xcf8fe6c55f319a95 };
```

```
uint64_t __k_cipher_range_257_512_const_0[8] = {
0x068b3e350cc445b3, 0x276805d874618924, 0x5f507e5b28605cbc, 0x5c27fd355e75b28d,
0x49b36eb3e4d23573, 0x4c73056b53772c7a, 0xab8c9fc4d78c0454, 0x8ce6bca6afec79f9 };
uint64_t __k_cipher_range_257_512_const_1[8] = {
0x5462124631ca6164, 0xa724c8137f74116f, 0x0f1354c2da386bd0, 0x366cea7f936a8239,
0xcd73a0329989e890, 0x094a21f6ac86924e, 0xc2e7059e2e236500, 0x3e2acb68917e5ddd };
uint64_t __k_cipher_range_257_512_const_2[8] = {
0x840d78c270b35588, 0x69fec54137a373ec, 0x2a5304a776d73039, 0xafdc8b13d985fefc,
0x6e9edd49afb69769, 0xe9a0562a218cdea6, 0xcfe78b3403a57159, 0xc0883f3842a0c83a };

uint64_t __k_cipher_range_513_1024_const_0[16] = {
0x2c846651558d88a5, 0x092441d00b6b2761, 0xf31bd4cef8a8c9bd, 0xf8db9bb403125ecc,
0x039407a206d7e350, 0x570f34e00ad05916, 0xbb9c486b0ce24273, 0x3232679a7682d1e3,
0x60954ef8305fb3c3, 0x89bb8d264b207631, 0x373560ec564f9107, 0x72c82368525c10f5,
0xa82671573cc0d8c9, 0x30badc85434799be, 0x61a6455b976f4e04, 0x5cd4d285ddef6853 };
uint64_t __k_cipher_range_513_1024_const_1[16] = {
0x6c2b8ec93b619cb3, 0x03e199ec0dfe4653, 0x5cfa0bc5348b7581, 0xbfd06463085993d4,
0x61d1ab62e76daf5d, 0xe25324516dc36709, 0x7282aa4339b46b1f, 0x8e80aa7c687caed9,
0x6691b63da3da2c28, 0xd774c4aecedd4e87, 0x17d807f35dc4fc02, 0xe6ef61514002414b,
0x8127857d776591fd, 0x14f6aa0d455cb8f7, 0xe3dee8fefc5a60d7, 0x0c732d6b745e3b99 };
uint64_t __k_cipher_range_513_1024_const_2[16] = {
0xe098d64e1591c952, 0xeed1ef778c15c711, 0x9cbe03931fe032d8, 0x6ca6995c96caf2bc,
0xdc3350e3d6ac261f, 0xe846910408a6070e, 0x2ae9efe7708677c1, 0x9a6c1cbbdd4c2734,
0xf18d8078bccb5fe5, 0xaad97acfc2796ab3, 0xa01b303d46469cd8, 0x54f10f2d2aabfe0a,
0xc785dc5071cb96b1, 0x055d780171ba4b4d, 0x3a5330d99edf77ed, 0x7da9b659017c0acd };
```

A.4 Substitution Box Widths

The following data structure stores the responses coming from the invocations to procedure GetSBoxLengths(). Responses are stored for all block lengths from $n \leftarrow 24$ to $n \leftarrow 1024$ in the following format: Each entry in the data structure stores the block length value n, the number of boxes b, the width of all boxes but the last m, a variable which is set to 0 if all boxes are of the same width, and to 1 if otherwise, and the last box width last, if different.

```
int
__k_cipher_all_mixing_box_length_data[1001][5] =
{
 { 24, 3, 8, 0, 0 },   { 25, 1, 25, 0, 0 },   { 26, 3, 8, 1, 10 },   { 27, 3, 9, 0, 0 },
 { 28, 2, 14, 0, 0 },   { 29, 4, 8, 1, 5 },   { 30, 3, 10, 0, 0 },   { 31, 4, 8, 1, 7 },
 { 32, 4, 8, 0, 0 },   { 33, 4, 8, 1, 9 },   { 34, 4, 8, 1, 10 },   { 35, 4, 9, 1, 8 },
 { 36, 4, 9, 0, 0 },   { 37, 5, 8, 1, 5 },   { 38, 5, 8, 1, 6 },   { 39, 5, 8, 1, 7 },
 { 40, 5, 8, 0, 0 },   { 41, 5, 8, 1, 9 },   { 42, 3, 14, 0, 0 },   { 43, 5, 9, 1, 7 },
 { 44, 5, 8, 1, 12 },   { 45, 5, 9, 0, 0 },   { 46, 6, 8, 1, 6 },   { 47, 6, 8, 1, 7 },
 { 48, 6, 8, 0, 0 },   { 49, 6, 8, 1, 9 },   { 50, 5, 10, 0, 0 },   { 51, 6, 9, 1, 6 },
 { 52, 6, 8, 1, 12 },   { 53, 6, 9, 1, 8 },   { 54, 6, 9, 0, 0 },   { 55, 7, 8, 1, 7 },
 { 56, 7, 8, 0, 0 },   { 57, 7, 8, 1, 9 },   { 58, 7, 8, 1, 10 },   { 59, 6, 9, 1, 14 },
 { 60, 6, 10, 0, 0 },   { 61, 7, 9, 1, 7 },   { 62, 7, 8, 1, 14 },   { 63, 7, 9, 0, 0 },
 { 64, 8, 8, 0, 0 },   { 65, 6, 10, 1, 15 },   { 66, 7, 9, 1, 12 },   { 67, 7, 10, 1, 7 },
 { 68, 7, 9, 1, 14 },   { 69, 7, 9, 1, 15 },   { 70, 7, 10, 0, 0 },   { 71, 8, 9, 1, 8 },
 { 72, 8, 9, 0, 0 },   { 73, 8, 9, 1, 10 },   { 74, 7, 10, 1, 14 },   { 75, 5, 15, 0, 0 },
 { 76, 7, 10, 1, 16 },   { 77, 8, 9, 1, 14 },   { 78, 8, 9, 1, 15 },   { 79, 8, 9, 1, 16 },
 { 80, 8, 10, 0, 0 },   { 81, 9, 9, 0, 0 },   { 82, 8, 10, 1, 12 },   { 83, 6, 15, 1, 8 },
 { 84, 7, 12, 0, 0 },   { 85, 8, 10, 1, 15 },   { 86, 8, 10, 1, 16 },   { 87, 7, 12, 1, 15 },
 { 88, 8, 10, 1, 18 },   { 89, 9, 10, 1, 9 },   { 90, 9, 10, 0, 0 },   { 91, 7, 14, 1, 7 },
 { 92, 9, 10, 1, 12 },   { 93, 8, 12, 1, 9 },   { 94, 9, 10, 1, 14 },   { 95, 9, 10, 1, 15 },
 { 96, 8, 12, 0, 0 },   { 97, 7, 15, 1, 7 },   { 98, 7, 14, 0, 0 },   { 99, 8, 12, 1, 15 },
 { 100, 10, 10, 0, 0 },   { 101, 6, 16, 1, 21 },   { 102, 8, 12, 1, 18 },   { 103, 7, 16, 1, 7 },
 { 104, 8, 12, 1, 20 },   { 105, 7, 15, 0, 0 },   { 106, 9, 12, 1, 10 },   { 107, 8, 14, 1, 9 },
 { 108, 9, 12, 0, 0 },   { 109, 6, 20, 1, 9 },   { 110, 9, 12, 1, 14 },   { 111, 9, 12, 1, 15 },
 { 112, 8, 14, 0, 0 },   { 113, 8, 14, 1, 15 },   { 114, 9, 12, 1, 18 },   { 115, 8, 15, 1, 10 },
 { 116, 9, 12, 1, 20 },   { 117, 9, 12, 1, 21 },   { 118, 10, 12, 1, 10 },   { 119, 8, 14, 1, 21 },
 { 120, 10, 12, 0, 0 },   { 121, 9, 14, 1, 9 },   { 122, 10, 12, 1, 14 },   { 123, 10, 12, 1, 15 },
 { 124, 10, 12, 1, 16 },   { 125, 5, 25, 0, 0 },   { 126, 9, 14, 0, 0 },   { 127, 9, 14, 1, 15 },
 { 128, 8, 16, 0, 0 },   { 129, 10, 12, 1, 21 },   { 130, 9, 14, 1, 18 },   { 131, 6, 25, 1, 6 },
 { 132, 11, 12, 0, 0 },   { 133, 9, 14, 1, 21 },   { 134, 11, 12, 1, 14 },   { 135, 9, 15, 0, 0 },
 { 136, 11, 12, 1, 16 },   { 137, 9, 16, 1, 9 },   { 138, 11, 12, 1, 18 },   { 139, 6, 25, 1, 14 },
```

```
{ 140, 10, 14, 0, 0 },  { 141, 11, 12, 1, 21 },  { 142, 10, 14, 1, 16 },  { 143, 9, 16, 1, 15 },
{ 144, 12, 12, 0, 0 },  { 145, 10, 15, 1, 10 },  { 146, 10, 14, 1, 20 },  { 147, 7, 21, 0, 0 },
{ 148, 9, 16, 1, 20 },  { 149, 10, 15, 1, 14 },  { 150, 10, 15, 0, 0 },  { 151, 10, 15, 1, 16 },
{ 152, 11, 14, 1, 12 },  { 153, 10, 15, 1, 18 },  { 154, 11, 14, 0, 0 },  { 155, 11, 14, 1, 15 },
{ 156, 11, 14, 1, 16 },  { 157, 8, 21, 1, 10 },  { 158, 11, 14, 1, 18 },  { 159, 10, 15, 1, 24 },
{ 160, 10, 16, 0, 0 },  { 161, 11, 14, 1, 21 },  { 162, 9, 18, 0, 0 },  { 163, 8, 21, 1, 16 },
{ 164, 11, 14, 1, 24 },  { 165, 11, 15, 0, 0 },  { 166, 12, 14, 1, 12 },  { 167, 8, 20, 1, 27 },
{ 168, 12, 14, 0, 0 },  { 169, 12, 14, 1, 15 },  { 170, 12, 14, 1, 16 },  { 171, 11, 15, 1, 21 },
{ 172, 12, 14, 1, 18 },  { 173, 6, 28, 1, 33 },  { 174, 12, 14, 1, 20 },  { 175, 7, 25, 0, 0 },
{ 176, 11, 16, 0, 0 },  { 177, 12, 15, 1, 12 },  { 178, 12, 14, 1, 24 },  { 179, 12, 14, 1, 25 },
{ 180, 12, 15, 0, 0 },  { 181, 12, 15, 1, 16 },  { 182, 13, 14, 0, 0 },  { 183, 13, 14, 1, 15 },
{ 184, 13, 14, 1, 16 },  { 185, 12, 15, 1, 20 },  { 186, 13, 14, 1, 18 },  { 187, 11, 16, 1, 27 },
{ 188, 13, 14, 1, 20 },  { 189, 9, 21, 0, 0 },  { 190, 12, 15, 1, 25 },  { 191, 12, 16, 1, 15 },
{ 192, 12, 16, 0, 0 },  { 193, 13, 14, 1, 25 },  { 194, 13, 15, 1, 14 },  { 195, 13, 15, 0, 0 },
{ 196, 14, 14, 0, 0 },  { 197, 12, 16, 1, 21 },  { 198, 11, 18, 0, 0 },  { 199, 10, 21, 1, 10 },
{ 200, 10, 20, 0, 0 },  { 201, 13, 15, 1, 21 },  { 202, 9, 24, 1, 10 },  { 203, 12, 16, 1, 27 },
{ 204, 13, 15, 1, 24 },  { 205, 13, 15, 1, 25 },  { 206, 13, 16, 1, 14 },  { 207, 13, 15, 1, 27 },
{ 208, 13, 16, 0, 0 },  { 209, 14, 15, 1, 14 },  { 210, 14, 15, 0, 0 },  { 211, 14, 15, 1, 16 },
{ 212, 13, 16, 1, 20 },  { 213, 14, 15, 1, 18 },  { 214, 12, 18, 1, 16 },  { 215, 14, 15, 1, 20 },
{ 216, 12, 18, 0, 0 },  { 217, 13, 16, 1, 25 },  { 218, 12, 18, 1, 20 },  { 219, 14, 15, 1, 24 },
{ 220, 11, 20, 0, 0 },  { 221, 11, 20, 1, 21 },  { 222, 14, 15, 1, 27 },  { 223, 14, 15, 1, 28 },
{ 224, 14, 16, 0, 0 },  { 225, 15, 15, 0, 0 },  { 226, 14, 16, 1, 18 },  { 227, 11, 20, 1, 27 },
{ 228, 14, 16, 1, 20 },  { 229, 14, 16, 1, 21 },  { 230, 13, 18, 1, 14 },  { 231, 11, 21, 0, 0 },
{ 232, 14, 16, 1, 24 },  { 233, 14, 16, 1, 25 },  { 234, 13, 18, 0, 0 },  { 235, 14, 16, 1, 27 },
{ 236, 14, 16, 1, 28 },  { 237, 13, 18, 1, 21 },  { 238, 14, 16, 1, 30 },  { 239, 15, 16, 1, 15 },
{ 240, 15, 16, 0, 0 },  { 241, 13, 18, 1, 25 },  { 242, 15, 16, 1, 18 },  { 243, 9, 27, 0, 0 },
{ 244, 15, 16, 1, 20 },  { 245, 15, 16, 1, 21 },  { 246, 13, 18, 1, 30 },  { 247, 12, 20, 1, 27 },
{ 248, 15, 16, 1, 24 },  { 249, 15, 16, 1, 25 },  { 250, 10, 25, 0, 0 },  { 251, 15, 16, 1, 27 },
{ 252, 14, 18, 0, 0 },  { 253, 10, 25, 1, 28 },  { 254, 15, 16, 1, 30 },  { 255, 14, 18, 1, 21 },
{ 256, 16, 16, 0, 0 },  { 257, 10, 25, 1, 32 },  { 258, 14, 18, 1, 24 },  { 259, 14, 18, 1, 25 },
{ 260, 13, 20, 0, 0 },  { 261, 14, 18, 1, 27 },  { 262, 14, 18, 1, 28 },  { 263, 12, 21, 1, 32 },
{ 264, 11, 24, 0, 0 },  { 265, 13, 20, 1, 25 },  { 266, 7, 38, 0, 0 },  { 267, 15, 18, 1, 15 },
{ 268, 15, 18, 1, 16 },  { 269, 8, 33, 1, 38 },  { 270, 15, 18, 0, 0 },  { 271, 11, 25, 1, 21 },
{ 272, 8, 34, 0, 0 },  { 273, 13, 21, 0, 0 },  { 274, 14, 20, 1, 14 },  { 275, 11, 25, 0, 0 },
{ 276, 15, 18, 1, 24 },  { 277, 15, 18, 1, 25 },  { 278, 14, 20, 1, 18 },  { 279, 15, 18, 1, 27 },
{ 280, 14, 20, 0, 0 },  { 281, 14, 20, 1, 21 },  { 282, 15, 18, 1, 30 },  { 283, 11, 25, 1, 33 },
{ 284, 15, 18, 1, 32 },  { 285, 15, 18, 1, 33 },  { 286, 16, 18, 1, 16 },  { 287, 14, 20, 1, 27 },
{ 288, 16, 18, 0, 0 },  { 289, 14, 21, 1, 16 },  { 290, 16, 18, 1, 20 },  { 291, 16, 18, 1, 21 },
{ 292, 14, 20, 1, 32 },  { 293, 14, 20, 1, 33 },  { 294, 14, 21, 0, 0 },  { 295, 16, 18, 1, 25 },
{ 296, 15, 20, 1, 16 },  { 297, 11, 27, 0, 0 },  { 298, 16, 18, 1, 28 },  { 299, 12, 25, 1, 24 },
{ 300, 15, 20, 0, 0 },  { 301, 15, 20, 1, 21 },  { 302, 16, 18, 1, 32 },  { 303, 16, 18, 1, 33 },
{ 304, 8, 38, 0, 0 },  { 305, 15, 20, 1, 25 },  { 306, 17, 18, 0, 0 },  { 307, 15, 20, 1, 27 },
{ 308, 11, 28, 0, 0 },  { 309, 17, 18, 1, 21 },  { 310, 15, 20, 1, 30 },  { 311, 12, 25, 1, 36 },
{ 312, 13, 24, 0, 0 },  { 313, 17, 18, 1, 25 },  { 314, 15, 20, 1, 34 },  { 315, 15, 21, 0, 0 },
{ 316, 17, 18, 1, 28 },  { 317, 12, 27, 1, 20 },  { 318, 17, 18, 1, 30 },  { 319, 15, 21, 1, 25 },
{ 320, 16, 20, 0, 0 },  { 321, 17, 18, 1, 33 },  { 322, 17, 18, 1, 34 },  { 323, 12, 28, 1, 15 },
{ 324, 18, 18, 0, 0 },  { 325, 13, 25, 0, 0 },  { 326, 15, 21, 1, 32 },  { 327, 16, 20, 1, 27 },
{ 328, 16, 20, 1, 28 },  { 329, 12, 27, 1, 32 },  { 330, 11, 30, 0, 0 },  { 331, 16, 21, 1, 16 },
{ 332, 16, 20, 1, 32 },  { 333, 16, 20, 1, 33 },  { 334, 16, 20, 1, 34 },  { 335, 16, 21, 1, 20 },
{ 336, 16, 21, 0, 0 },  { 337, 14, 24, 1, 25 },  { 338, 17, 20, 1, 18 },  { 339, 16, 21, 1, 24 },
{ 340, 17, 20, 0, 0 },  { 341, 17, 20, 1, 21 },  { 342, 9, 38, 0, 0 },  { 343, 7, 49, 0, 0 },
{ 344, 17, 20, 1, 24 },  { 345, 17, 20, 1, 25 },  { 346, 14, 24, 1, 34 },  { 347, 17, 20, 1, 27 },
{ 348, 17, 20, 1, 28 },  { 349, 16, 21, 1, 34 },  { 350, 14, 25, 0, 0 },  { 351, 13, 27, 0, 0 },
{ 352, 11, 32, 0, 0 },  { 353, 17, 20, 1, 33 },  { 354, 17, 20, 1, 34 },  { 355, 14, 25, 1, 30 },
{ 356, 17, 20, 1, 36 },  { 357, 17, 21, 0, 0 },  { 358, 18, 20, 1, 18 },  { 359, 14, 25, 1, 34 },
{ 360, 18, 20, 0, 0 },  { 361, 18, 20, 1, 21 },  { 362, 13, 27, 1, 38 },  { 363, 11, 33, 0, 0 },
{ 364, 13, 28, 0, 0 },  { 365, 18, 20, 1, 25 },  { 366, 17, 21, 1, 30 },  { 367, 18, 20, 1, 27 },
{ 368, 18, 20, 1, 28 },  { 369, 17, 21, 1, 33 },  { 370, 18, 20, 1, 30 },  { 371, 15, 25, 1, 21 },
{ 372, 18, 20, 1, 32 },  { 373, 18, 20, 1, 33 },  { 374, 11, 34, 0, 0 },  { 375, 15, 25, 0, 0 },
{ 376, 18, 20, 1, 36 },  { 377, 18, 21, 1, 20 },  { 378, 18, 21, 0, 0 },  { 379, 14, 27, 1, 28 },
{ 380, 19, 20, 0, 0 },  { 381, 19, 20, 1, 21 },  { 382, 18, 21, 1, 25 },  { 383, 15, 25, 1, 33 },
{ 384, 16, 24, 0, 0 },  { 385, 7, 55, 0, 0 },  { 386, 15, 25, 1, 36 },  { 387, 19, 20, 1, 27 },
{ 388, 19, 20, 1, 28 },  { 389, 18, 21, 1, 32 },  { 390, 13, 30, 0, 0 },  { 391, 18, 21, 1, 34 },
{ 392, 14, 28, 0, 0 },  { 393, 19, 20, 1, 33 },  { 394, 19, 20, 1, 34 },  { 395, 18, 21, 1, 38 },
{ 396, 12, 33, 0, 0 },  { 397, 14, 28, 1, 33 },  { 398, 19, 20, 1, 38 },  { 399, 19, 21, 0, 0 },
{ 400, 20, 20, 0, 0 },  { 401, 12, 33, 1, 38 },  { 402, 19, 21, 1, 24 },  { 403, 19, 21, 1, 25 },
{ 404, 17, 24, 1, 20 },  { 405, 15, 27, 0, 0 },  { 406, 19, 21, 1, 28 },  { 407, 16, 25, 1, 32 },
{ 408, 17, 24, 0, 0 },  { 409, 17, 24, 1, 25 },  { 410, 19, 21, 1, 32 },  { 411, 19, 21, 1, 33 },
{ 412, 19, 21, 1, 34 },  { 413, 16, 25, 1, 38 },  { 414, 19, 21, 1, 36 },  { 415, 16, 25, 1, 40 },
```

```
{ 416, 13, 32, 0, 0 },  { 417, 19, 21, 1, 39 },  { 418, 11, 38, 0, 0 },  { 419, 20, 21, 1, 20 },
{ 420, 20, 21, 0, 0 },  { 421, 17, 25, 1, 21 },  { 422, 17, 24, 1, 38 },  { 423, 20, 21, 1, 24 },
{ 424, 20, 21, 1, 25 },  { 425, 17, 25, 0, 0 },  { 426, 20, 21, 1, 27 },  { 427, 20, 21, 1, 28 },
{ 428, 18, 24, 1, 20 },  { 429, 13, 33, 0, 0 },  { 430, 17, 25, 1, 30 },  { 431, 20, 21, 1, 32 },
{ 432, 18, 24, 0, 0 },  { 433, 20, 21, 1, 34 },  { 434, 17, 25, 1, 34 },  { 435, 20, 21, 1, 36 },
{ 436, 18, 24, 1, 28 },  { 437, 20, 21, 1, 38 },  { 438, 20, 21, 1, 39 },  { 439, 20, 21, 1, 40 },
{ 440, 11, 40, 0, 0 },  { 441, 21, 21, 0, 0 },  { 442, 13, 34, 0, 0 },  { 443, 18, 25, 1, 18 },
{ 444, 18, 24, 1, 36 },  { 445, 18, 25, 1, 20 },  { 446, 18, 24, 1, 38 },  { 447, 18, 24, 1, 39 },
{ 448, 16, 28, 0, 0 },  { 449, 18, 25, 1, 24 },  { 450, 18, 25, 0, 0 },  { 451, 12, 38, 1, 33 },
{ 452, 19, 24, 1, 20 },  { 453, 19, 24, 1, 21 },  { 454, 16, 28, 1, 34 },  { 455, 18, 25, 1, 30 },
{ 456, 19, 24, 0, 0 },  { 457, 19, 24, 1, 25 },  { 458, 18, 25, 1, 33 },  { 459, 17, 27, 0, 0 },
{ 460, 19, 24, 1, 28 },  { 461, 18, 25, 1, 36 },  { 462, 14, 33, 0, 0 },  { 463, 18, 25, 1, 38 },
{ 464, 19, 24, 1, 32 },  { 465, 19, 24, 1, 33 },  { 466, 19, 24, 1, 34 },  { 467, 18, 25, 1, 42 },
{ 468, 13, 36, 0, 0 },  { 469, 17, 28, 1, 21 },  { 470, 19, 24, 1, 38 },  { 471, 19, 24, 1, 39 },
{ 472, 19, 24, 1, 40 },  { 473, 17, 28, 1, 25 },  { 474, 19, 24, 1, 42 },  { 475, 19, 25, 0, 0 },
{ 476, 17, 28, 0, 0 },  { 477, 20, 24, 1, 21 },  { 478, 19, 25, 1, 28 },  { 479, 18, 27, 1, 20 },
{ 480, 20, 24, 0, 0 },  { 481, 20, 24, 1, 25 },  { 482, 19, 25, 1, 32 },  { 483, 20, 24, 1, 27 },
{ 484, 11, 44, 0, 0 },  { 485, 12, 40, 1, 45 },  { 486, 18, 27, 0, 0 },  { 487, 18, 27, 1, 28 },
{ 488, 20, 24, 1, 32 },  { 489, 20, 24, 1, 33 },  { 490, 10, 49, 0, 0 },  { 491, 18, 27, 1, 32 },
{ 492, 20, 24, 1, 36 },  { 493, 18, 27, 1, 34 },  { 494, 13, 38, 0, 0 },  { 495, 15, 33, 0, 0 },
{ 496, 20, 24, 1, 40 },  { 497, 18, 27, 1, 38 },  { 498, 20, 24, 1, 42 },  { 499, 20, 25, 1, 24 },
{ 500, 20, 25, 0, 0 },  { 501, 21, 24, 1, 21 },  { 502, 20, 25, 1, 27 },  { 503, 20, 25, 1, 28 },
{ 504, 21, 24, 0, 0 },  { 505, 21, 24, 1, 25 },  { 506, 19, 27, 1, 20 },  { 507, 13, 39, 0, 0 },
{ 508, 21, 24, 1, 28 },  { 509, 20, 25, 1, 34 },  { 510, 17, 30, 0, 0 },  { 511, 20, 25, 1, 36 },
{ 512, 16, 32, 0, 0 },  { 513, 19, 27, 0, 0 },  { 514, 21, 24, 1, 34 },  { 515, 20, 25, 1, 40 },
{ 516, 21, 24, 1, 36 },  { 517, 20, 25, 1, 42 },  { 518, 21, 24, 1, 38 },  { 519, 21, 24, 1, 39 },
{ 520, 13, 40, 0, 0 },  { 521, 21, 25, 1, 21 },  { 522, 21, 24, 1, 42 },  { 523, 16, 33, 1, 28 },
{ 524, 21, 24, 1, 44 },  { 525, 21, 25, 0, 0 },  { 526, 19, 27, 1, 40 },  { 527, 21, 25, 1, 27 },
{ 528, 22, 24, 0, 0 },  { 529, 22, 24, 1, 25 },  { 530, 21, 25, 1, 30 },  { 531, 22, 24, 1, 27 },
{ 532, 19, 28, 0, 0 },  { 533, 21, 25, 1, 33 },  { 534, 22, 24, 1, 30 },  { 535, 18, 30, 1, 25 },
{ 536, 22, 24, 1, 32 },  { 537, 22, 24, 1, 33 },  { 538, 22, 24, 1, 34 },  { 539, 11, 49, 0, 0 },
{ 540, 20, 27, 0, 0 },  { 541, 20, 27, 1, 28 },  { 542, 22, 24, 1, 38 },  { 543, 22, 24, 1, 39 },
{ 544, 17, 32, 0, 0 },  { 545, 21, 25, 1, 45 },  { 546, 14, 39, 0, 0 },  { 547, 20, 27, 1, 34 },
{ 548, 22, 24, 1, 44 },  { 549, 22, 24, 1, 45 },  { 550, 22, 25, 0, 0 },  { 551, 20, 27, 1, 38 },
{ 552, 23, 24, 0, 0 },  { 553, 23, 24, 1, 25 },  { 554, 18, 30, 1, 44 },  { 555, 23, 24, 1, 27 },
{ 556, 23, 24, 1, 28 },  { 557, 22, 25, 1, 32 },  { 558, 23, 24, 1, 30 },  { 559, 22, 25, 1, 34 },
{ 560, 20, 28, 0, 0 },  { 561, 17, 33, 0, 0 },  { 562, 23, 24, 1, 34 },  { 563, 22, 25, 1, 38 },
{ 564, 23, 24, 1, 36 },  { 565, 22, 25, 1, 40 },  { 566, 23, 24, 1, 38 },  { 567, 21, 27, 0, 0 },
{ 568, 23, 24, 1, 40 },  { 569, 22, 25, 1, 44 },  { 570, 19, 30, 0, 0 },  { 571, 20, 28, 1, 39 },
{ 572, 13, 44, 0, 0 },  { 573, 23, 24, 1, 45 },  { 574, 23, 25, 1, 24 },  { 575, 23, 25, 0, 0 },
{ 576, 24, 24, 0, 0 },  { 577, 23, 25, 1, 27 },  { 578, 17, 34, 0, 0 },  { 579, 21, 27, 1, 39 },
{ 580, 23, 25, 1, 30 },  { 581, 21, 28, 1, 21 },  { 582, 23, 25, 1, 32 },  { 583, 23, 25, 1, 33 },
{ 584, 23, 25, 1, 34 },  { 585, 15, 39, 0, 0 },  { 586, 23, 25, 1, 36 },  { 587, 21, 28, 1, 27 },
{ 588, 21, 28, 0, 0 },  { 589, 23, 25, 1, 39 },  { 590, 23, 25, 1, 40 },  { 591, 22, 27, 1, 24 },
{ 592, 23, 25, 1, 42 },  { 593, 21, 28, 1, 33 },  { 594, 22, 27, 0, 0 },  { 595, 23, 25, 1, 45 },
{ 596, 21, 28, 1, 36 },  { 597, 22, 27, 1, 30 },  { 598, 23, 25, 1, 48 },  { 599, 24, 25, 1, 24 },
{ 600, 24, 25, 0, 0 },  { 601, 22, 27, 1, 34 },  { 602, 24, 25, 1, 27 },  { 603, 24, 25, 1, 28 },
{ 604, 21, 28, 1, 44 },  { 605, 11, 55, 0, 0 },  { 606, 22, 27, 1, 39 },  { 607, 24, 25, 1, 32 },
{ 608, 19, 32, 0, 0 },  { 609, 24, 25, 1, 34 },  { 610, 20, 30, 1, 40 },  { 611, 24, 25, 1, 36 },
{ 612, 18, 34, 0, 0 },  { 613, 24, 25, 1, 38 },  { 614, 24, 25, 1, 39 },  { 615, 24, 25, 1, 40 },
{ 616, 22, 28, 0, 0 },  { 617, 24, 25, 1, 42 },  { 618, 23, 27, 1, 24 },  { 619, 24, 25, 1, 44 },
{ 620, 24, 25, 1, 45 },  { 621, 23, 27, 0, 0 },  { 622, 23, 27, 1, 28 },  { 623, 24, 25, 1, 48 },
{ 624, 16, 39, 0, 0 },  { 625, 25, 25, 0, 0 },  { 626, 23, 27, 1, 32 },  { 627, 19, 33, 0, 0 },
{ 628, 23, 27, 1, 34 },  { 629, 20, 32, 1, 21 },  { 630, 21, 30, 0, 0 },  { 631, 15, 44, 1, 15 },
{ 632, 23, 27, 1, 38 },  { 633, 23, 27, 1, 39 },  { 634, 23, 27, 1, 40 },  { 635, 20, 32, 1, 27 },
{ 636, 23, 27, 1, 42 },  { 637, 13, 49, 0, 0 },  { 638, 23, 27, 1, 44 },  { 639, 23, 27, 1, 45 },
{ 640, 20, 32, 0, 0 },  { 641, 23, 28, 1, 25 },  { 642, 23, 27, 1, 48 },  { 643, 23, 27, 1, 49 },
{ 644, 23, 28, 0, 0 },  { 645, 24, 27, 1, 24 },  { 646, 19, 34, 0, 0 },  { 647, 20, 32, 1, 39 },
{ 648, 24, 27, 0, 0 },  { 649, 24, 27, 1, 28 },  { 650, 13, 50, 0, 0 },  { 651, 24, 27, 1, 30 },
{ 652, 23, 28, 1, 36 },  { 653, 24, 27, 1, 32 },  { 654, 24, 27, 1, 33 },  { 655, 24, 27, 1, 34 },
{ 656, 23, 28, 1, 40 },  { 657, 24, 27, 1, 36 },  { 658, 23, 28, 1, 42 },  { 659, 24, 27, 1, 38 },
{ 660, 22, 30, 0, 0 },  { 661, 24, 27, 1, 40 },  { 662, 22, 30, 1, 32 },  { 663, 17, 39, 0, 0 },
{ 664, 23, 28, 1, 48 },  { 665, 24, 27, 1, 44 },  { 666, 24, 27, 1, 45 },  { 667, 21, 32, 1, 27 },
{ 668, 24, 28, 1, 24 },  { 669, 24, 27, 1, 48 },  { 670, 24, 27, 1, 49 },  { 671, 24, 27, 1, 50 },
{ 672, 24, 28, 0, 0 },  { 673, 25, 27, 1, 25 },  { 674, 24, 28, 1, 30 },  { 675, 25, 27, 0, 0 },
{ 676, 13, 52, 0, 0 },  { 677, 24, 28, 1, 33 },  { 678, 25, 27, 1, 30 },  { 679, 22, 30, 1, 49 },
{ 680, 20, 34, 0, 0 },  { 681, 25, 27, 1, 33 },  { 682, 25, 27, 1, 34 },  { 683, 24, 28, 1, 39 },
{ 684, 19, 36, 0, 0 },  { 685, 23, 30, 1, 25 },  { 686, 14, 49, 0, 0 },  { 687, 25, 27, 1, 39 },
{ 688, 25, 27, 1, 40 },  { 689, 24, 28, 1, 45 },  { 690, 23, 30, 0, 0 },  { 691, 20, 34, 1, 45 },
```

```
{ 692, 25, 27, 1, 44 },  { 693, 21, 33, 0, 0 },  { 694, 24, 28, 1, 50 },  { 695, 20, 34, 1, 49 },
{ 696, 25, 27, 1, 48 },  { 697, 25, 27, 1, 49 },  { 698, 25, 27, 1, 50 },  { 699, 25, 28, 1, 27 },
{ 700, 25, 28, 0, 0 },  { 701, 21, 34, 1, 21 },  { 702, 26, 27, 0, 0 },  { 703, 26, 27, 1, 28 },
{ 704, 22, 32, 0, 0 },  { 705, 26, 27, 1, 30 },  { 706, 25, 28, 1, 34 },  { 707, 26, 27, 1, 32 },
{ 708, 26, 27, 1, 33 },  { 709, 26, 27, 1, 34 },  { 710, 25, 28, 1, 38 },  { 711, 26, 27, 1, 36 },
{ 712, 25, 28, 1, 40 },  { 713, 26, 27, 1, 38 },  { 714, 21, 34, 0, 0 },  { 715, 13, 55, 0, 0 },
{ 716, 25, 28, 1, 44 },  { 717, 26, 27, 1, 42 },  { 718, 24, 30, 1, 28 },  { 719, 26, 27, 1, 44 },
{ 720, 24, 30, 0, 0 },  { 721, 25, 28, 1, 49 },  { 722, 19, 38, 0, 0 },  { 723, 26, 27, 1, 48 },
{ 724, 26, 27, 1, 49 },  { 725, 26, 27, 1, 50 },  { 726, 22, 33, 0, 0 },  { 727, 26, 27, 1, 52 },
{ 728, 26, 28, 0, 0 },  { 729, 27, 27, 0, 0 },  { 730, 26, 28, 1, 30 },  { 731, 23, 32, 1, 27 },
{ 732, 26, 28, 1, 32 },  { 733, 26, 28, 1, 33 },  { 734, 26, 28, 1, 34 },  { 735, 15, 49, 0, 0 },
{ 736, 23, 32, 0, 0 },  { 737, 23, 32, 1, 33 },  { 738, 26, 28, 1, 38 },  { 739, 26, 28, 1, 39 },
{ 740, 26, 28, 1, 40 },  { 741, 19, 39, 0, 0 },  { 742, 26, 28, 1, 42 },  { 743, 23, 32, 1, 39 },
{ 744, 26, 28, 1, 44 },  { 745, 26, 28, 1, 45 },  { 746, 23, 32, 1, 42 },  { 747, 25, 30, 1, 27 },
{ 748, 22, 34, 0, 0 },  { 749, 26, 28, 1, 49 },  { 750, 25, 30, 0, 0 },  { 751, 23, 33, 1, 25 },
{ 752, 26, 28, 1, 52 },  { 753, 25, 30, 1, 33 },  { 754, 26, 28, 1, 54 },  { 755, 27, 28, 1, 27 },
{ 756, 27, 28, 0, 0 },  { 757, 19, 39, 1, 55 },  { 758, 27, 28, 1, 30 },  { 759, 23, 33, 0, 0 },
{ 760, 20, 38, 0, 0 },  { 761, 27, 28, 1, 33 },  { 762, 27, 28, 1, 34 },  { 763, 24, 32, 1, 27 },
{ 764, 27, 28, 1, 36 },  { 765, 17, 45, 0, 0 },  { 766, 27, 28, 1, 38 },  { 767, 27, 28, 1, 39 },
{ 768, 24, 32, 0, 0 },  { 769, 25, 30, 1, 49 },  { 770, 14, 55, 0, 0 },  { 771, 23, 33, 1, 45 },
{ 772, 27, 28, 1, 44 },  { 773, 27, 28, 1, 45 },  { 774, 25, 30, 1, 54 },  { 775, 25, 30, 1, 55 },
{ 776, 27, 28, 1, 48 },  { 777, 27, 28, 1, 49 },  { 778, 27, 28, 1, 50 },  { 779, 20, 39, 1, 38 },
{ 780, 26, 30, 0, 0 },  { 781, 24, 32, 1, 45 },  { 782, 23, 34, 0, 0 },  { 783, 27, 28, 1, 55 },
{ 784, 28, 28, 0, 0 },  { 785, 24, 32, 1, 49 },  { 786, 26, 30, 1, 36 },  { 787, 24, 33, 1, 28 },
{ 788, 26, 30, 1, 38 },  { 789, 26, 30, 1, 39 },  { 790, 26, 30, 1, 40 },  { 791, 24, 32, 1, 55 },
{ 792, 24, 33, 0, 0 },  { 793, 25, 32, 1, 25 },  { 794, 26, 30, 1, 44 },  { 795, 26, 30, 1, 45 },
{ 796, 25, 32, 1, 28 },  { 797, 24, 33, 1, 38 },  { 798, 21, 38, 0, 0 },  { 799, 26, 30, 1, 49 },
{ 800, 25, 32, 0, 0 },  { 801, 25, 32, 1, 33 },  { 802, 26, 30, 1, 52 },  { 803, 24, 33, 1, 44 },
{ 804, 26, 30, 1, 54 },  { 805, 26, 30, 1, 55 },  { 806, 26, 30, 1, 56 },  { 807, 27, 30, 1, 27 },
{ 808, 27, 30, 1, 28 },  { 809, 24, 33, 1, 50 },  { 810, 27, 30, 0, 0 },  { 811, 24, 33, 1, 52 },
{ 812, 27, 30, 1, 32 },  { 813, 27, 30, 1, 33 },  { 814, 27, 30, 1, 34 },  { 815, 24, 33, 1, 56 },
{ 816, 24, 34, 0, 0 },  { 817, 25, 32, 1, 49 },  { 818, 27, 30, 1, 38 },  { 819, 21, 39, 0, 0 },
{ 820, 27, 30, 1, 40 },  { 821, 24, 34, 1, 39 },  { 822, 27, 30, 1, 42 },  { 823, 25, 32, 1, 55 },
{ 824, 27, 30, 1, 44 },  { 825, 25, 33, 0, 0 },  { 826, 25, 33, 1, 34 },  { 827, 26, 32, 1, 27 },
{ 828, 23, 36, 0, 0 },  { 829, 27, 30, 1, 49 },  { 830, 27, 30, 1, 50 },  { 831, 25, 33, 1, 39 },
{ 832, 26, 32, 0, 0 },  { 833, 17, 49, 0, 0 },  { 834, 27, 30, 1, 54 },  { 835, 27, 30, 1, 55 },
{ 836, 22, 38, 0, 0 },  { 837, 25, 33, 1, 45 },  { 838, 28, 30, 1, 28 },  { 839, 26, 32, 1, 39 },
{ 840, 28, 30, 0, 0 },  { 841, 25, 33, 1, 49 },  { 842, 28, 30, 1, 32 },  { 843, 28, 30, 1, 33 },
{ 844, 28, 30, 1, 34 },  { 845, 26, 32, 1, 45 },  { 846, 28, 30, 1, 36 },  { 847, 25, 33, 1, 55 },
{ 848, 28, 30, 1, 38 },  { 849, 28, 30, 1, 39 },  { 850, 25, 34, 0, 0 },  { 851, 22, 39, 1, 32 },
{ 852, 28, 30, 1, 42 },  { 853, 26, 33, 1, 28 },  { 854, 28, 30, 1, 44 },  { 855, 19, 45, 0, 0 },
{ 856, 26, 32, 1, 56 },  { 857, 26, 33, 1, 32 },  { 858, 26, 33, 0, 0 },  { 859, 28, 30, 1, 49 },
{ 860, 28, 30, 1, 50 },  { 861, 26, 33, 1, 36 },  { 862, 28, 30, 1, 52 },  { 863, 26, 33, 1, 38 },
{ 864, 27, 32, 0, 0 },  { 865, 28, 30, 1, 55 },  { 866, 28, 30, 1, 56 },  { 867, 26, 33, 1, 42 },
{ 868, 27, 32, 1, 36 },  { 869, 26, 33, 1, 44 },  { 870, 29, 30, 0, 0 },  { 871, 27, 32, 1, 39 },
{ 872, 29, 30, 1, 32 },  { 873, 29, 30, 1, 33 },  { 874, 23, 38, 0, 0 },  { 875, 26, 33, 1, 50 },
{ 876, 29, 30, 1, 36 },  { 877, 27, 32, 1, 45 },  { 878, 29, 30, 1, 38 },  { 879, 29, 30, 1, 39 },
{ 880, 22, 40, 0, 0 },  { 881, 27, 32, 1, 49 },  { 882, 21, 42, 0, 0 },  { 883, 26, 34, 1, 33 },
{ 884, 26, 34, 0, 0 },  { 885, 29, 30, 1, 45 },  { 886, 27, 32, 1, 54 },  { 887, 27, 32, 1, 55 },
{ 888, 29, 30, 1, 48 },  { 889, 29, 30, 1, 49 },  { 890, 29, 30, 1, 50 },  { 891, 27, 33, 0, 0 },
{ 892, 29, 30, 1, 52 },  { 893, 20, 45, 1, 38 },  { 894, 29, 30, 1, 54 },  { 895, 29, 30, 1, 55 },
{ 896, 28, 32, 0, 0 },  { 897, 23, 39, 0, 0 },  { 898, 28, 32, 1, 34 },  { 899, 26, 34, 1, 49 },
{ 900, 30, 30, 0, 0 },  { 901, 24, 38, 1, 27 },  { 902, 28, 32, 1, 38 },  { 903, 28, 32, 1, 39 },
{ 904, 28, 32, 1, 40 },  { 905, 26, 34, 1, 55 },  { 906, 28, 32, 1, 42 },  { 907, 27, 33, 1, 49 },
{ 908, 28, 32, 1, 44 },  { 909, 28, 32, 1, 45 },  { 910, 27, 33, 1, 52 },  { 911, 27, 34, 1, 27 },
{ 912, 24, 38, 0, 0 },  { 913, 28, 32, 1, 49 },  { 914, 28, 32, 1, 50 },  { 915, 22, 42, 1, 33 },
{ 916, 28, 32, 1, 52 },  { 917, 27, 34, 1, 33 },  { 918, 27, 34, 0, 0 },  { 919, 28, 32, 1, 55 },
{ 920, 23, 40, 0, 0 },  { 921, 28, 33, 1, 30 },  { 922, 27, 34, 1, 38 },  { 923, 28, 33, 1, 32 },
{ 924, 28, 33, 0, 0 },  { 925, 28, 33, 1, 34 },  { 926, 29, 32, 1, 30 },  { 927, 28, 33, 1, 36 },
{ 928, 29, 32, 0, 0 },  { 929, 29, 32, 1, 33 },  { 930, 29, 32, 1, 34 },  { 931, 19, 49, 0, 0 },
{ 932, 29, 32, 1, 36 },  { 933, 28, 33, 1, 42 },  { 934, 29, 32, 1, 38 },  { 935, 17, 55, 0, 0 },
{ 936, 26, 36, 0, 0 },  { 937, 25, 38, 1, 25 },  { 938, 29, 32, 1, 42 },  { 939, 28, 33, 1, 48 },
{ 940, 29, 32, 1, 44 },  { 941, 29, 32, 1, 45 },  { 942, 26, 36, 1, 42 },  { 943, 28, 33, 1, 52 },
{ 944, 29, 32, 1, 48 },  { 945, 21, 45, 0, 0 },  { 946, 29, 32, 1, 50 },  { 947, 28, 33, 1, 56 },
{ 948, 29, 32, 1, 52 },  { 949, 26, 36, 1, 49 },  { 950, 25, 38, 0, 0 },  { 951, 29, 32, 1, 55 },
{ 952, 28, 34, 0, 0 },  { 953, 24, 39, 1, 56 },  { 954, 29, 33, 1, 30 },  { 955, 26, 36, 1, 55 },
{ 956, 29, 32, 1, 60 },  { 957, 29, 33, 0, 0 },  { 958, 30, 32, 1, 30 },  { 959, 24, 40, 1, 39 },
{ 960, 30, 32, 0, 0 },  { 961, 30, 32, 1, 33 },  { 962, 30, 32, 1, 34 },  { 963, 29, 33, 1, 39 },
{ 964, 30, 32, 1, 36 },  { 965, 24, 40, 1, 45 },  { 966, 23, 42, 0, 0 },  { 967, 30, 32, 1, 39 },
```

```
 { 968, 22, 44, 0, 0 },   { 969, 29, 33, 1, 45 },   { 970, 30, 32, 1, 42 },   { 971, 20, 49, 1, 40 },
 { 972, 27, 36, 0, 0 },   { 973, 30, 32, 1, 45 },   { 974, 29, 33, 1, 50 },   { 975, 25, 39, 0, 0 },
 { 976, 30, 32, 1, 48 },   { 977, 30, 32, 1, 49 },   { 978, 30, 32, 1, 50 },   { 979, 29, 33, 1, 55 },
 { 980, 20, 49, 0, 0 },   { 981, 27, 36, 1, 45 },   { 982, 30, 32, 1, 54 },   { 983, 30, 32, 1, 55 },
 { 984, 30, 32, 1, 56 },   { 985, 29, 34, 1, 33 },   { 986, 29, 34, 0, 0 },   { 987, 30, 33, 1, 30 },
 { 988, 26, 38, 0, 0 },   { 989, 30, 33, 1, 32 },   { 990, 30, 33, 0, 0 },   { 991, 30, 33, 1, 34 },
 { 992, 31, 32, 0, 0 },   { 993, 31, 32, 1, 33 },   { 994, 31, 32, 1, 34 },   { 995, 30, 33, 1, 38 },
 { 996, 31, 32, 1, 36 },   { 997, 30, 33, 1, 40 },   { 998, 31, 32, 1, 38 },   { 999, 31, 32, 1, 39 },
 { 1000, 25, 40, 0, 0 },   { 1001, 30, 33, 1, 44 },   { 1002, 31, 32, 1, 42 },   { 1003, 26, 39, 1, 28 },
 { 1004, 31, 32, 1, 44 },   { 1005, 31, 32, 1, 45 },   { 1006, 30, 33, 1, 49 },   { 1007, 30, 33, 1, 50},
 { 1008, 28, 36, 0, 0 },   { 1009, 31, 32, 1, 49 },   { 1010, 31, 32, 1, 50 },   { 1011, 30, 33, 1, 54 },
 { 1012, 23, 44, 0, 0 },   { 1013, 30, 33, 1, 56 },   { 1014, 26, 39, 0, 0 },   { 1015, 31, 32, 1, 55 },
 { 1016, 31, 32, 1, 56 },   { 1017, 30, 33, 1, 60 },   { 1018, 30, 34, 1, 32 },   { 1019, 30, 34, 1, 33},
 { 1020, 30, 34, 0, 0 },   { 1021, 28, 36, 1, 49 },   { 1022, 31, 33, 1, 32 },   { 1023, 31, 33, 0, 0 },
 { 1024, 32, 32, 0, 0 }
};
```

A.5 Generating Pseudorandom Index Sequences for Bit Reordering

We have used the following routine to implement the assignment $\stackrel{\$}{\leftarrow}$ Permutation(), which is used by procedure BitReorderingIndexes():

```
compute_reordering_indexes(int indexes [], int n)
 {
     int i = 0, m = 0, u = 0, v = 0;
     for (i = 0; i < 10; i++)
     {
        v = (int)((double)n * (double)rand() / (double)RAND_MAX);
     }
     for (i = 0; i < n; i++)
     {
        indexes[i] = -1;
     }
     v = n;
     while (v >= n)
     {
        v = (int)((double)n * (double)rand() / (double)RAND_MAX);
     }
     indexes[n - 1] = v;
     u = v;
     for (i = 0; i < n - 1; i++)
     {
       if (i == u)
           continue;
        bool placed = false;
        do
        {
           v = n;
           while (v >= n)
           {
              v = (int)((double)n * (double)rand() / (double)RAND_MAX);
           }
           if (indexes[v] < 0)
           {
              indexes[v] = i;
              placed = true;
           }
        } while (placed == false);
     }
     for (i = 0; i < n; i++)
     {
        if (indexes[i] < 0)
        {
           indexes[i] = n - 1;
        }
     }
```

```
    return 1;
}
```

We have also used the standard C library call `rand()` to implement the assignment $\xleftarrow{\$}$ Range(), which is found in line 19 of procedure BitReorderingIndexes(). This is done in the following way:

```
val = (int)((double)rangeval * (double)rand() / (double)RAND_MAX);
```

Using the default seed for `rand()` we produced index sequences for all block lengths in the range [24, 1024]. The following index sequences are used by some block lengths:

```
int _k_cipher_24_bit_reordering_0[24] = {
7, 4, 14, 22, 0, 11, 18, 9, 6, 20, 1, 21, 10, 15, 3, 8,
2, 16, 5, 19, 12, 13, 17, 23};

int _k_cipher_24_bit_reordering_1[24] = {
1, 19, 7, 10, 16, 21, 15, 2, 5, 13, 18, 12, 23, 8, 17, 4,
9, 22, 0, 3, 11, 6, 14, 20};

int _k_cipher_24_bit_reordering_2[24] = {
22, 17, 2, 14, 7, 10, 9, 20, 6, 3, 16, 21, 11, 15, 0, 18,
4, 12, 5, 1, 8, 13, 23, 19};

int _k_cipher_24_bit_reordering_3[24] = {
7, 16, 12, 1, 2, 13, 20, 23, 14, 19, 21, 6, 9, 4, 0, 11,
5, 3, 17, 18, 15, 22, 10, 8};

//index sequences 4, 5, 6 and 7 are not used by the block length n <- 24

int _k_cipher_24_bit_reordering_8[24] = {
23, 15, 18, 5, 8, 0, 3, 12, 10, 19, 21, 7, 16, 9, 13, 4,
20, 11, 6, 1, 17, 2, 22, 14};

int _k_cipher_24_bit_reordering_9[24] = {
12, 3, 19, 16, 22, 13, 1, 5, 4, 8, 7, 21, 17, 11, 15, 18,
14, 23, 10, 6, 2, 20, 0, 9};

int _k_cipher_24_bit_reordering_10[24] = {
4, 10, 16, 14, 1, 18, 8, 0, 15, 7, 12, 5, 20, 21, 2, 13,
17, 22, 6, 19, 9, 11, 3, 23};
int _k_cipher_24_bit_reordering_11[24] = {
18, 0, 7, 19, 15, 8, 21, 2, 13, 16, 3, 20, 11, 9, 22, 6,
4, 14, 10, 1, 23, 5, 17, 12};

int _k_cipher_24_bit_reordering_12[24] = {
14, 19, 2, 9, 16, 18, 8, 4, 20, 6, 5, 12, 17, 21, 3, 13,
10, 1, 15, 23, 7, 11, 0, 22};

int _k_cipher_24_bit_reordering_13[24] = {
14, 3, 4, 17, 13, 16, 11, 0, 23, 12, 22, 15, 2, 5, 8, 20,
1, 18, 19, 9, 6, 10, 21, 7};

int _k_cipher_33_bit_reordering_0[33] = {
27, 3, 29, 7, 13, 0, 9, 20, 23, 15, 8, 2, 30, 26, 11, 24,
17, 1, 28, 31, 16, 18, 21, 10, 5, 12, 14, 19, 4, 6, 32, 25,
22};

int _k_cipher_33_bit_reordering_1[33] = {
17, 27, 5, 2, 10, 31, 24, 0, 15, 13, 21, 29, 28, 12, 4, 6,
19, 7, 3, 9, 25, 23, 18, 14, 30, 11, 26, 8, 32, 22, 1, 16,
20};

int _k_cipher_33_bit_reordering_2[33] = {
18, 15, 21, 32, 4, 28, 10, 0, 5, 8, 26, 11, 29, 3, 23, 17,
14, 2, 16, 24, 12, 19, 30, 27, 7, 6, 25, 9, 13, 1, 22, 31,
20};
```

```
int __k_cipher_33_bit_reordering_3[33] = {
0, 3, 11, 16, 23, 30, 6, 27, 17, 8, 15, 32, 4, 18, 9, 26,
24, 12, 19, 31, 22, 5, 2, 28, 14, 29, 13, 21, 7, 25, 10, 20,
1};

//index sequences 4, 5, 6 and 7 are not used by the block length n <- 33

int __k_cipher_33_bit_reordering_8[33] = {
20, 14, 25, 23, 12, 1, 0, 30, 8, 2, 27, 16, 11, 7, 22, 29,
18, 3, 32, 9, 13, 21, 6, 28, 17, 10, 4, 5, 19, 15, 24, 26,
31};

int __k_cipher_33_bit_reordering_9[33] = {
8, 18, 21, 25, 14, 29, 0, 2, 12, 7, 9, 27, 4, 16, 32, 17,
22, 20, 10, 24, 28, 15, 5, 3, 30, 26, 11, 31, 23, 19, 6, 13,
1};

int __k_cipher_33_bit_reordering_10[33] = {
5, 17, 11, 1, 28, 24, 29, 3, 10, 6, 23, 14, 25, 4, 26, 9,
20, 16, 21, 27, 7, 22, 32, 8, 15, 31, 13, 0, 18, 2, 12, 19,
30};

int __k_cipher_33_bit_reordering_11[33] = {
7, 30, 3, 18, 14, 2, 15, 17, 27, 19, 4, 25, 13, 9, 23, 8,
31, 0, 22, 16, 32, 10, 29, 21, 6, 20, 26, 1, 12, 11, 24, 5,
28};

int __k_cipher_33_bit_reordering_12[33] = {
7, 29, 17, 13, 4, 8, 25, 24, 9, 27, 6, 11, 20, 28, 16, 1,
18, 15, 0, 21, 32, 2, 30, 14, 19, 26, 10, 23, 5, 12, 22, 31,
3};

int __k_cipher_33_bit_reordering_13[33] = {
0, 32, 22, 1, 12, 21, 6, 28, 9, 14, 30, 2, 17, 26, 24, 10,
3, 8, 13, 18, 31, 27, 20, 4, 16, 29, 15, 7, 23, 25, 5, 19,
11};

int __k_cipher_128_bit_reordering_0[128] = {
79, 121, 89, 83, 99, 31, 64, 21, 40, 62, 7, 33, 9, 114, 111, 54,
4, 68, 73, 90, 103, 110, 37, 115, 41, 52, 84, 124, 8, 16, 28, 63,
10, 96, 93, 32, 50, 2, 43, 109, 86, 125, 60, 119, 29, 66, 74, 18,
53, 20, 97, 42, 81, 12, 1, 106, 30, 70, 35, 59, 118, 78, 120, 95,
108, 6, 47, 58, 116, 23, 11, 82, 127, 65, 26, 36, 48, 91, 72, 100,
46, 104, 56, 80, 0, 27, 76, 51, 67, 22, 117, 38, 126, 92, 98, 13,
17, 101, 88, 123, 57, 45, 15, 75, 107, 5, 69, 49, 113, 39, 85, 25,
3, 122, 44, 77, 55, 105, 24, 112, 61, 14, 87, 102, 34, 19, 71, 94};

int __k_cipher_128_bit_reordering_1[128] = {
73, 33, 14, 45, 95, 83, 1, 110, 52, 124, 16, 99, 115, 27, 64, 56,
2, 71, 92, 126, 39, 101, 12, 53, 109, 116, 77, 41, 26, 22, 59, 82,
63, 36, 105, 18, 121, 86, 11, 118, 46, 75, 66, 5, 89, 54, 103, 29,
125, 3, 47, 81, 76, 15, 51, 102, 114, 28, 88, 108, 21, 57, 67, 38,
20, 34, 84, 106, 113, 42, 91, 120, 98, 78, 55, 62, 7, 68, 8, 24,
79, 65, 10, 31, 60, 122, 6, 49, 43, 32, 96, 85, 90, 23, 107, 119,
100, 40, 72, 112, 58, 4, 123, 93, 13, 111, 50, 19, 37, 70, 25, 80,
44, 74, 17, 117, 87, 30, 94, 9, 69, 35, 48, 0, 61, 127, 97, 104};

int __k_cipher_128_bit_reordering_2[128] = {
106, 116, 18, 98, 70, 7, 95, 125, 50, 9, 85, 59, 26, 45, 78, 34,
111, 83, 94, 39, 4, 127, 48, 68, 75, 96, 28, 56, 20, 114, 8, 42,
122, 101, 47, 63, 115, 6, 77, 29, 36, 17, 107, 87, 51, 11, 65, 93,
104, 55, 124, 37, 46, 119, 91, 103, 13, 64, 79, 16, 30, 82, 61, 0,
120, 84, 71, 23, 27, 118, 88, 57, 32, 15, 108, 52, 102, 40, 1, 72,
19, 66, 92, 24, 117, 60, 10, 105, 123, 5, 49, 81, 33, 76, 41, 100,
109, 89, 31, 112, 69, 97, 126, 43, 54, 14, 74, 86, 2, 58, 21, 35,
73, 62, 12, 113, 3, 121, 110, 44, 99, 38, 67, 90, 80, 22, 25, 53};

int __k_cipher_128_bit_reordering_3[128] = {
```

```
25, 65, 83, 115, 76, 48, 10, 3, 47, 18, 110, 126, 98, 56, 90, 38,
99, 24, 40, 72, 6, 84, 89, 69, 20, 54, 59, 33, 112, 12, 106, 120,
34, 15, 46, 2, 16, 70, 101, 82, 61, 50, 118, 88, 79, 121, 104, 29,
0, 62, 22, 114, 26, 75, 8, 123, 44, 107, 52, 71, 93, 100, 37, 86,
103, 51, 109, 39, 11, 57, 119, 64, 95, 30, 42, 19, 73, 125, 1, 81,
9, 49, 87, 23, 91, 45, 7, 97, 68, 32, 28, 124, 63, 108, 78, 117,
27, 80, 17, 77, 122, 41, 14, 5, 35, 113, 67, 55, 102, 92, 105, 60,
36, 74, 96, 127, 43, 94, 4, 85, 21, 13, 111, 31, 66, 116, 58, 53};

int __k_cipher_128_bit_reordering_4[128] = {
20, 92, 2, 104, 59, 67, 8, 100, 37, 85, 49, 113, 77, 121, 44, 28,
110, 6, 10, 71, 56, 93, 118, 32, 55, 16, 29, 82, 43, 98, 76, 126,
124, 66, 99, 83, 17, 25, 88, 48, 39, 3, 12, 60, 47, 75, 112, 111,
87, 90, 4, 33, 13, 46, 68, 24, 54, 105, 18, 123, 117, 103, 78, 57,
122, 26, 53, 19, 64, 34, 116, 89, 81, 62, 107, 79, 102, 45, 14, 5,
15, 114, 36, 61, 72, 95, 86, 109, 7, 65, 120, 96, 41, 31, 23, 52,
80, 11, 101, 70, 21, 106, 119, 42, 73, 0, 35, 51, 58, 91, 30, 127,
74, 94, 1, 125, 97, 38, 27, 63, 9, 115, 50, 84, 108, 22, 40, 69};

int __k_cipher_128_bit_reordering_5[128] = {
28, 34, 78, 58, 106, 118, 86, 100, 95, 46, 6, 65, 11, 19, 127, 50,
107, 89, 122, 16, 25, 51, 81, 32, 42, 97, 8, 69, 116, 72, 1, 62,
48, 68, 111, 87, 119, 39, 20, 124, 102, 59, 91, 44, 24, 15, 5, 77,
103, 12, 75, 33, 83, 105, 21, 43, 90, 52, 27, 117, 7, 66, 63, 126,
55, 112, 64, 120, 56, 110, 47, 101, 10, 35, 4, 26, 18, 85, 76, 94,
22, 61, 41, 3, 73, 30, 98, 125, 93, 84, 113, 13, 70, 53, 104, 36,
80, 115, 92, 17, 79, 54, 123, 31, 37, 9, 40, 57, 2, 96, 71, 109,
74, 38, 67, 121, 60, 99, 14, 88, 29, 108, 82, 49, 0, 114, 23, 45};

int __k_cipher_128_bit_reordering_6[128] = {
64, 85, 78, 43, 37, 118, 122, 107, 12, 30, 100, 1, 18, 51, 94, 63,
3, 10, 23, 95, 62, 81, 40, 50, 36, 99, 66, 79, 105, 119, 27, 125,
90, 117, 38, 14, 5, 57, 55, 87, 73, 111, 20, 123, 25, 103, 44, 69,
17, 32, 126, 97, 28, 13, 106, 114, 58, 42, 88, 54, 80, 6, 75, 71,
101, 0, 39, 124, 41, 29, 84, 8, 70, 76, 109, 48, 113, 59, 92, 22,
31, 112, 108, 53, 33, 47, 82, 65, 4, 60, 16, 91, 98, 121, 11, 77,
15, 115, 24, 67, 74, 2, 86, 52, 93, 19, 127, 110, 35, 56, 102, 46,
72, 26, 61, 21, 7, 96, 45, 9, 49, 34, 120, 116, 104, 83, 89, 68};

int __k_cipher_128_bit_reordering_7[128] = {
94, 84, 8, 115, 61, 48, 0, 106, 29, 23, 127, 71, 98, 44, 75, 37,
19, 3, 25, 66, 99, 125, 40, 112, 52, 107, 15, 88, 33, 80, 57, 78,
51, 83, 104, 58, 123, 38, 103, 93, 67, 6, 30, 43, 20, 13, 113, 73,
74, 124, 105, 101, 85, 70, 34, 92, 1, 11, 116, 56, 31, 42, 22, 55,
4, 47, 54, 86, 90, 114, 17, 12, 72, 122, 26, 100, 59, 32, 68, 108,
2, 41, 62, 69, 110, 121, 16, 81, 14, 27, 102, 77, 118, 39, 50, 91,
87, 36, 5, 89, 96, 49, 60, 45, 28, 9, 18, 120, 64, 109, 117, 79,
21, 119, 111, 46, 63, 53, 82, 126, 76, 35, 97, 7, 24, 10, 95, 65};

int __k_cipher_128_bit_reordering_8[128] = {
63, 69, 122, 29, 40, 96, 117, 89, 54, 6, 87, 13, 21, 73, 106, 34,
119, 36, 30, 91, 52, 107, 67, 76, 124, 16, 84, 14, 44, 100, 2, 58,
26, 56, 0, 126, 83, 105, 112, 70, 55, 23, 47, 93, 75, 102, 15, 39,
98, 113, 1, 85, 71, 50, 45, 78, 111, 61, 120, 95, 32, 8, 18, 24,
90, 77, 33, 49, 114, 66, 80, 103, 20, 62, 3, 31, 11, 121, 41, 110,
127, 101, 17, 51, 42, 9, 86, 64, 104, 5, 60, 92, 35, 118, 28, 72,
65, 53, 79, 97, 88, 12, 7, 109, 25, 37, 115, 22, 57, 125, 46, 82,
74, 123, 99, 38, 94, 27, 48, 116, 68, 81, 4, 10, 59, 108, 19, 43};

int __k_cipher_128_bit_reordering_9[128] = {
27, 108, 119, 45, 72, 21, 4, 87, 58, 15, 127, 97, 92, 54, 35, 67,
107, 125, 24, 39, 98, 90, 47, 7, 73, 83, 68, 16, 53, 63, 118, 10,
106, 66, 20, 86, 8, 116, 51, 6, 100, 25, 94, 78, 60, 124, 42, 34,
102, 74, 59, 52, 5, 115, 43, 29, 65, 12, 88, 37, 122, 85, 22, 105,
80, 19, 89, 9, 123, 61, 70, 99, 109, 77, 26, 49, 2, 33, 117, 41,
103, 30, 55, 79, 17, 14, 56, 36, 104, 112, 126, 81, 46, 71, 3, 93,
31, 69, 91, 62, 76, 40, 114, 48, 110, 82, 13, 120, 1, 38, 101, 23,
84, 28, 0, 113, 32, 96, 75, 57, 64, 50, 18, 11, 121, 95, 111, 44};
```

```
int __k_cipher_128_bit_reordering_10[128] = {
84, 54, 37, 112, 16, 105, 65, 10, 28, 12, 32, 70, 53, 95, 121, 102,
29, 96, 47, 125, 49, 7, 89, 69, 118, 111, 74, 85, 30, 44, 56, 5,
35, 11, 124, 58, 75, 22, 91, 109, 8, 24, 51, 38, 114, 101, 80, 66,
76, 107, 36, 87, 25, 48, 15, 116, 82, 100, 67, 59, 42, 120, 9, 31,
6, 73, 45, 88, 17, 106, 57, 126, 78, 18, 46, 103, 86, 115, 61, 0,
83, 52, 71, 3, 26, 110, 40, 122, 98, 2, 19, 77, 93, 34, 127, 63,
33, 50, 94, 4, 79, 97, 123, 20, 81, 117, 55, 104, 64, 39, 21, 14,
119, 108, 13, 23, 68, 90, 60, 43, 62, 1, 113, 99, 27, 41, 92, 72};

int __k_cipher_128_bit_reordering_11[128] = {
123, 6, 16, 49, 101, 43, 86, 76, 78, 119, 82, 38, 22, 104, 2, 53,
10, 114, 35, 107, 64, 60, 29, 93, 79, 110, 28, 13, 57, 47, 117, 83,
89, 1, 65, 121, 33, 108, 63, 20, 97, 27, 69, 88, 112, 3, 40, 50,
122, 87, 106, 54, 8, 23, 45, 74, 15, 61, 100, 30, 84, 124, 75, 32,
14, 81, 42, 62, 77, 120, 109, 17, 98, 0, 113, 41, 52, 26, 73, 80,
111, 51, 31, 5, 66, 91, 37, 116, 58, 44, 92, 70, 18, 103, 118, 4,
90, 126, 72, 11, 96, 21, 55, 46, 127, 34, 67, 94, 59, 24, 7, 105,
99, 68, 56, 12, 25, 115, 39, 95, 71, 36, 85, 102, 9, 48, 19, 125};

int __k_cipher_128_bit_reordering_12[128] = {
63, 78, 108, 116, 20, 89, 37, 5, 30, 9, 86, 45, 114, 56, 105, 73,
59, 41, 2, 80, 28, 110, 125, 67, 83, 126, 12, 68, 26, 39, 60, 98,
72, 92, 15, 111, 40, 51, 121, 19, 77, 94, 31, 103, 119, 13, 52, 34,
22, 90, 8, 44, 75, 127, 104, 49, 27, 71, 109, 11, 85, 62, 113, 35,
57, 46, 81, 122, 23, 100, 4, 66, 79, 112, 106, 24, 93, 38, 14, 58,
124, 91, 61, 17, 65, 10, 107, 43, 70, 97, 123, 54, 82, 47, 18, 6,
25, 101, 3, 120, 95, 33, 76, 55, 48, 87, 0, 42, 74, 96, 118, 16,
99, 115, 29, 36, 1, 84, 69, 53, 64, 117, 32, 88, 50, 7, 102, 21};

int __k_cipher_128_bit_reordering_13[128] = {
48, 78, 35, 7, 118, 103, 20, 86, 54, 80, 6, 68, 29, 121, 102, 33,
36, 98, 9, 75, 24, 120, 50, 83, 17, 0, 52, 96, 90, 47, 73, 123,
89, 27, 32, 104, 112, 62, 15, 67, 18, 101, 74, 116, 56, 85, 34, 8,
5, 81, 41, 65, 58, 127, 25, 107, 13, 69, 126, 26, 111, 40, 49, 92,
71, 1, 124, 106, 88, 23, 37, 59, 19, 76, 113, 53, 4, 99, 94, 44,
97, 79, 39, 2, 21, 119, 63, 82, 43, 22, 14, 84, 109, 60, 117, 72,
114, 87, 12, 16, 61, 38, 108, 64, 46, 110, 30, 57, 93, 66, 10, 122,
28, 105, 51, 3, 125, 95, 42, 70, 31, 45, 100, 55, 91, 77, 11, 115};
```

A.6 **Flex** Flow Test Vectors

```
uint32_t __k_cipher_24_flex_plain = 0x318f00;
uint32_t __k_cipher_24_flex_tweak = 0x5c1703;
uint32_t __k_cipher_24_flex_key[3] = {0x597a95ce, 0x2cbed1e4, 0x4d82b5db};
uint32_t __k_cipher_24_flex_cipher = 0xd89875;

uint64_t __k_cipher_33_flex_plain = 0x071fc2a25;
uint64_t __k_cipher_33_flex_tweak = 0x0f3b9df4c;
uint64_t __k_cipher_33_flex_key[2] = {0x2a9a999187600201, 0x84ccc79a0e5972a9};
uint64_t __k_cipher_33_flex_cipher = 0x11db7d054;

uint64_t __k_cipher_128_flex_plain[2] = {0xd7c635db3c752489, 0x06d64b43649bfefe};
uint64_t __k_cipher_128_flex_tweak[2] = {0x9bbb62981bbdf274, 0xfc83cc6c39265c91};
uint64_t __k_cipher_128_flex_key[2] = {0x4546ea020eac175f, 0x77bd347bb9d5b095};
uint64_t __k_cipher_128_flex_cipher[2] = {0xd2a60c5b43ccd3e1, 0xca718842e3900a4a};
```

A.7 **CPA** Flow Test Vectors

```
uint32_t __k_cipher_24_cpa_plain = 0x318f00;
uint32_t __k_cipher_24_cpa_tweak = 0x9a0e59;
uint64_t __k_cipher_24_cpa_key[4] = {0x2cbed1e4597a95ce, 0x255c17034d82b5db, 0x876002011e71fc2a,
0x000072a92a9a9991};
uint32_t __k_cipher_24_cpa_cipher = 0x9ebd08;

uint64_t __k_cipher_33_cpa_plain = 0x14c84ccc7;
```

```
uint64_t __k_cipher_33_cpa_tweak = 0x05c919bbb;
uint64_t __k_cipher_33_cpa_key[6] = {0xdb3c752489ecf3b9, 0x43649bfefed7c635, 0x020eac175f06d64b,
0x7bb9d5b0954546ea, 0x981bbdf27477bd34, 0x0000000000000022};
uint64_t __k_cipher_33_cpa_cipher = 0x09467b3f7;

uint64_t __k_cipher_128_cpa_plain[2] = {0x2b58bffc83cc6c39, 0x24f22580b2107da7};
uint64_t __k_cipher_128_cpa_tweak[2] = {0xe40ced6f189f18fe, 0x739aa03368c43949};
uint64_t __k_cipher_128_cpa_key[14] = {0x27aef6116c4db0e6, 0x2779d02d3094d1df, 0xb8c0ad914767ba80,
0x6ca98308d45d1f79, 0xd75f78588ceaf21a, 0x3190bc4bfa457450,
0x92fd07e27f65d6c2, 0xd632a79fd631870c, 0x235548ef50bd1c1f,
0x002440be99b4d4ba, 0x1d038d1d35d9cd0f, 0xb1336f128aaebf73,
0x8028a087933b6f4a, 0x74fd2d5530ebb1f5};
uint64_t __k_cipher_128_cpa_cipher[2] = {0xa3f53f715d01e6eb, 0xa8c1904ee7567837};
```